Vorwort

C++ ist keine Sprache für Programmieranfänger. Höchstwahrscheinlich besitzen Sie schon Grundkenntnisse in einer höheren Programmiersprache. Dann möchten Sie nicht mit ausführlichen Erklärungen der elementaren Konzepte höherer Programmiersprachen gelangweilt werden, sondern erfahren, worin sich C++ von der Ihnen bekannten Sprache unterscheidet.

Dieses Buch wendet sich vor allem an Leser, die bereits in Pascal programmiert haben. Die Sprachkonstruktionen, die sich in C++ kaum von Pascal unterscheiden, werden nur ganz knapp gegenübergestellt. So bleibt genügend Raum, die wirklich entscheidenden Neuerungen in C++ darzustellen und vor besonderen Fallstricken für Pascal-Programmierer zu warnen. Natürlich genügt es nicht, Pascal-Formulierungen in C++ umsetzen zu können. Sie werden vor allem auch in die Denkweise der objektorientierten Programmierung eingeführt.

Das Buch besteht aus drei großen Teilbereichen: Die Kapitel 1 und 2 dienen dem Umstieg von der Pascal-Programmierung zur konventionellen Programmierung in C++. Hier genügen knappe Programmbeispiele, die die vorgestellten Sprachmittel auf möglichst engem Raum klären.

Das dritte Kapitel führt Sie in die objektorientierte Programmierung in C++ ein. Danach sollten Sie diese neue Denkweise und die wichtigsten Sprachmittel von C++ im Prinzip beherrschen. Einige weniger wichtige Konzepte wurden bewußt vernachlässigt. Hier wurde einer soliden Beherrschung der wichtigen Konzepte der Vorzug vor Vollständigkeit gegeben. Zweifelsfälle werden vor allem dann behandelt, wenn sie zu schwer auffindbaren Programmfehlern führen können, nicht aber, wenn sie eindeutig durch den Compiler erkannt und erklärt werden.

Nun verhält es sich aber mit Programmiersprachen - und ganz besonders mit so komplexen Sprachen wie C++ - ähnlich wie mit dem Schachspiel: Die Kenntnis der Regeln macht noch keinen Schachspieler. Entscheidend ist die Fähigkeit, die erlernten Regeln in praktisches Handeln umzusetzen. Diese Fähigkeit kann man sich nicht allein anlesen. Sie muß durch eigenes Tun erworben werden. Der Weg läßt sich aber deutlich verkürzen, wenn man an den Erfahrungen anderer teilhaben kann.

Das vierte Kapitel bietet Ihnen deshalb die Möglichkeit, Ihre neu erworbenen Kenntnisse im praktischen Einsatz zu vertiefen. Der Nutzen objektorientierter Methoden erschließt sich vor allem bei größeren Programmierprojekten. Deshalb stelle ich Ihnen in diesem letzten Kapitel des Buchs Projekte vor, in denen sinnvoll einsetzbare Programme entwickelt werden. Es ist eine Erfahrungsregel, daß erst ein Bruchteil der Arbeit getan ist, wenn ein Programm "im Prinzip fertig" ist. Den Hauptteil der Arbeit machen dann solche "Kleinigkeiten" aus, wie eine komfortable Benutzeroberfläche, Robustheit gegen

Fehlbedienung oder kleine zusätzliche Verbesserungen der Funktion (der Appetit kommt mit dem Essen). Sie finden zu jedem Projekt eine leicht verständliche Prototyp-Version mit ausführlichen Erklärungen. Umfangreichere komfortable Versionen mit weiterer Dokumentation finden Sie auf der dem Buch beiliegenden Diskette. Unter diesen Programmen befinden sich interessante Spiel- und Simulationsprogramme, aber auch praktisch einsetzbare Programme, wie ein komfortables Zeichenprogramm für Graphenstrukturen. Dazu erhalten Sie ein komplettes Entwicklungspaket für Programme unter einer komfortablen grafischen Benutzeroberfläche für alle gängigen Grafikkarten. Was Sie in der Hand halten, müßte man daher eigentlich als Diskette mit beiliegendem Buch bezeichnen.

Alle Programmbeispiele in diesem Buch und auf der beiliegenden Diskette sind unter Borland C++ lauffähig. Solange keine Programme für Microsoft Windows entwickelt werden, gibt es keine wesentlichen Unterschiede zwischen Borland C++ und Turbo C++. Alles, was in diesem Buch über Borland C++ gesagt wird, gilt auch für Turbo C++. Für Anwender von Zortech C++ enthält die Diskette Hinweise zur Portierung. Viele Programme, so auch die grafische Benutzeroberfläche, verwenden keine Borland-spezifischen Funktionen und sind unverändert auch unter Zortech C++ einsetzbar.

1 Von Pascal nach C++

Man hört oft die Meinung, C sei eine schwierige Programmiersprache. Da C++ eine Erweiterung von C ist (jedes C-Programm ist auch ein C++-Programm!), gilt dies umso mehr auch für C++. Wenn Sie von Pascal nach C++ umsteigen wollen, haben Sie vor allem wohl folgende Hürden zu überwinden:

- Zunächst gibt es einige sehr harmlose Unterschiede, nämlich Sprachmittel, die in C++ im Prinzip wie in Pascal, aber mit anderen Symbolen ausgedrückt werden. Zum Beispiel wird der logische Operator `or` aus Pascal in C++ durch `||` dargestellt. Hier könnte man also bei der gewohnten Schreibweise bleiben und vor dem Übersetzen eines Programmes mit einem Texteditor überall `or` durch `||` ersetzen. Diese Umsetzung kann der Compiler sogar selbst mit Hilfe des sogenannten Präprozessors übernehmen. Wenn Ihnen das hilft, können Sie für eine Übergangszeit bei einigen besonders ungewöhnlichen Operatoren die vertraute Pascal-Schreibweise beibehalten.
- Die Art der Bezeichnung von Zeigertypen, insbesondere in höheren Stufen, ist nicht ganz einfach zu durchschauen. Hier lassen sich Anfangshürden vermeiden, indem jeder verwendete Zeigertyp zunächst in einer Typdeklaration mit einem Namen versehen wird.
- C++ läßt Ihnen als Programmierer viel mehr Freiheiten als etwa Pascal oder gar Modula-2. Insbesondere erlaubt C++ (verlangt es aber nicht!) einen extrem kompakten Programmierstil, der für Ungeübte nur sehr schwer zu durchschauen ist. Die verschiedenen Möglichkeiten zur Darstellung der gleichen Situation führen dazu, daß jeder Programmierer seinen individuellen Stil entwickelt, der oft von anderen schwer nachvollzogen werden kann. In dieser Einführung verzichte ich bewußt auf einige Möglichkeiten der Sprache und empfehle Ihnen einen einfachen

Programmierstil, der es Ihnen ermöglicht, wenigstens Ihre eigenen Programme später wieder zu durchschauen.

Was die Schwierigkeit einer Programmiersprache betrifft, so sollten Sie unterscheiden zwischen der Schwierigkeit, die Sprache zu erlernen, und der Schwierigkeit, Probleme mit der Sprache zu lösen. Ein Bagger ist zwar schwieriger zu bedienen als ein Suppenlöffel. Beim Ausheben einer Baugrube werden Sie sich aber mit dem Bagger leichter tun. In diesem Sinne ist C++ ein leistungsfähiger Bagger, über dessen Design und Ergonomie man sicher geteilter Meinung sein kann.

Dieses einführende Kapitel soll Ihnen den Umstieg von Pascal nach C++ möglichst leicht machen. Sie werden deshalb hauptsächlich die Sprachmittel kennenlernen, die es in gleicher oder ähnlicher Form auch in Pascal (oder im erweiterten Sprachvorrat von Turbo Pascal) gibt. Wenn Sie sich am Ende dieses Kapitels sich noch nicht für C++ erwärmen können, liegt das einfach daran, daß Sie die Sprachmittel, die den eigentlichen Wert von C++ ausmachen, erst später kennenlernen werden. Sie werden in diesem Kapitel vergeblich nach umfangreicheren, realistischen Programmbeispielen suchen. Dafür gibt es zwei Gründe. Zum einen sind Sie ja bereits mit Pascal (oder vielleicht Modula-2) vertraut, so daß knappe Gegenüberstellungen ausreichen dürften, um Ihnen die Unterschiede in der Syntax zu verdeutlichen. Zum anderen möchte ich die wirklich interessanten Beispielprogramme solange zurückstellen, bis wir sie mit objektorientierten Methoden angehen können.

1.1 Programmstruktur

Nach diesen Vorbemerkungen wollen wir an einem ersten Beispiel einige wichtige Unterschiede zwischen Pascal und C++ betrachten. Das folgende Programm liest einen String ein, wandelt alle Kleinbuchstaben in Großbuchstaben um und prüft, ob die vorkommenden runden Klammern paarweise passend angeordnet sind. So erzeugt etwa die Eingabe "`a*(b+c)`" die Ausgabe "`richtig: A*(B+C)`". Das ganze soll kein sinnvolles Programm sein, sondern nur auf engem Raum möglichst viele typische elementare Sprachelemente vorführen.

Bevor Sie beim Vergleich der beiden Programmversionen C++ in Grund und Boden verdammen, bedenken Sie bitte, daß es sich hier um einen ungleichen Wettbewerb handelt: Sie sehen auf der rechten Seite gewissermaßen ein C++ mit angezogener Handbremse. Hier kommt keine der Stärken von C++ zum Tragen. So ist es kein Wunder, wenn Pascal hier wegen seiner lesbareren Syntax noch das Rennen macht.

```
program Klammern;                      /* Klammern */
                                       #include <string.h>
                                       #include <ctype.h>
                                       #include <iostream.h>

  const n = 80;                        const int n = 80;

  type  Zeile = string[n];             typedef char Zeile[n+1];

var   s: Zeile;                        Zeile s;
```

```
   function korrekt (s: Zeile):
                      boolean;
   (* prüft Zeichenkette auf
15    korrekte Klammerstruktur *)
     var i, offen: integer;
     begin
       i := 1;
       offen := 0;

20     while (i <= Length (s))
         and (offen >= 0) do begin

         if s[i] = '(' then
           offen := offen + 1

         else if s[i] = ')' then
25         offen := offen -1;
         i := i + 1;
       end;
       korrekt := offen = 0;
     end;

30
   procedure bearbeite (var s:
                             Zeile);
   (* verwandelt alle Buchstaben
      in Großbuchstaben *)
35   var i: integer;
     begin
       for i := 1 to Length (s) do
         case s[i] of
           'ä': s[i] := 'Ä';
40         'ö': s[i] := 'Ö';
           'ü': s[i] := 'Ü';
         else s[i] := UpCase(s[i])
         end;
     end;
45

   begin
     repeat
       writeln;
       write('Ausdruck: ');
50     readln (s);
       bearbeite (s);
       if korrekt (s) then
         write ('richtig: ')
       else write ('falsch : ');
55     writeln (s);

     until s = 'ENDE'
   end.
```

```
int korrekt (Zeile s)
/* prüft eine Zeichenkette
   auf korrekte Klammerstruktur */
{
  int i, offen;

  i = 0;
  offen = 0;

  while (i < strlen (s)
    && offen >= 0) {

    if (s[i] == '(')
      offen = offen + 1;

    else if (s[i] == ')')
      offen = offen - 1;
    i = i + 1;
  }
  return offen == 0;
}


void bearbeite (Zeile s)
  /* verwandelt alle Buchstaben
      in Großbuchstaben */
{
  int i;

  for (i=0; i<strlen (s); i=i+1)
    switch (s[i]) {
      case 'ä': s[i] = 'Ä'; break;
      case 'ö': s[i] = 'Ö'; break;
      case 'ü': s[i] = 'Ü'; break;
      default: s[i]=toupper(s[i]);
    }
}


main () {
  do {
    cout << "\nAusdruck: ";

    cin >> s;
    bearbeite (s);
    if (korrekt (s))
      cout << "richtig: ";
    else cout << "falsch : ";
    cout << s << '\n';
  }
  while (strcmp (s, "ENDE") != 0);
}
```

Sie können und sollen hier noch nicht alle Details der rechten Spalte völlig verstehen. Es geht einfach darum, daß Sie eine gewisse Vorstellung von der syntaktischen Form eines C++-Programms bekommen. Auf einige Besonderheiten möchte ich hier schon eingehen. Sie werden sie alle später noch einmal systematisch kennenlernen.

- *Zeile 1:* Das Schlüsselwort `program` entfällt in C++ ersatzlos. Stattdessen sehen Sie rechts einen Kommentar.
- *Zeilen 2-4:* Diese Zeilen entsprechen den `uses`-Anweisungen in Turbo Pascal (Die Pascal-Version benötigt keine speziellen Units).

- *Zeile 6:* In C++ gibt es nur typisierte Konstanten.
- *Zeile 8:* In C++ gibt es keinen speziellen String-Typ. Die Deklaration entspricht `array[0..n] of char`.
- *Zeile 10:* Variablen werden in C++ ohne ein spezielles Schlüsselwort deklariert. Erst kommt der Typname und dann der (oder die) Bezeichner.
- *Zeile 12:* Auch Funktionsdefinitionen kommen in C++ ohne ein eigenes Schlüsselwort aus. Der Typ des Funktionswerts steht am Anfang. Es gibt keinen eigenen Typ `boolean`.
- *Zeilen 16-17:* Die Schlüsselwörter `begin` und `end` werden in C++ durch geschweifte Klammern ersetzt. Lokale Deklarationen stehen in C++ innerhalb der Blockklammern.
- *Zeilen 18-19:* Für die Zuweisung verwendet C++ das einfache Gleichheitszeichen. In Pascal werden Strings ab eins indiziert, in C++ dagegen ab Null.
- *Zeile 20-21:* Bei der `while`-Anweisung entfällt das Schlüsselwort `do`. Dafür muß die Bedingung in runde Klammern eingeschlossen werden. Die logische Und-Verknüpfung wird in C++ durch `&&` ausgedrückt. Die Klammerung der beiden Teilaussagen kann in C++ entfallen, weil es dort differenziertere Vorrangregeln gibt.
- *Zeile 22:* Die `if`-Anweisung verhält sich wie die `while`-Anweisung. Hier entfällt das Schlüsselwort `then`. Die Bedingung muß ebenfalls in runde Klammern eingeschlossen werden. Für die Gleichheitsrelation wird in C++ ein doppeltes Gleichheitszeichen verwendet.
- *Zeilen 23-24:* In Pascal darf vor `else` kein Semikolon stehen, in C++ muß es stehen. Das liegt daran, daß in C++ jede Anweisung mit einem Semikolon endet, in Pascal dagegen das Semikolon nur zwischen zwei Anweisungen steht (Vor dem `else` endet zwar eine Anweisung; mit dem `else` beginnt aber keine neue Anweisung, sondern nur der `else`-Zweig der umfassenden Anweisung!).
- *Zeile 28:* Der Funktionswert wird in C++ mit der `return`-Anweisung (wie in Modula-2) zurückgegeben.
- *Zeile 31:* C++ macht keinen Unterschied zwischen Funktionen und Prozeduren. Eine Prozedur ist formal eine Funktion mit leerem Wertebereich (`void`).
- *Zeile 37:* Die `for`-Anweisung hat in C++ eine wesentlich flexiblere, aber auch schwerer lesbare Struktur der Form:
  `for (`**Anfangsaktion Fortsetzungsbedingung** `;` **Inkrementierungsaktion**`)`
  Anweisung`;`
- *Zeilen 38-43:* Auch die `case`-Anweisung sieht in C++ etwas anders aus. Statt `case` steht in C++ `switch`, das Wort `case` wird in C++ den einzelnen Selektoren vorangestellt.
- *Zeile 46:* In C++ ist das Hauptprogramm formal eine Funktion mit dem speziellen Namen `main`.
- *Zeile 47:* Aus der `repeat`-Schleife wird in C++ eine `do`-Schleife mit zwei wichtigen Unterschieden: Erstens darf nach dem `do` nur eine Anweisung stehen. Zweitens prüft C++ nicht wie Pascal eine Abbruchbedingung, sondern das logische Gegenteil, nämlich eine Fortsetzungsbedingung.
- *Zeile 48-49:* `cout` steht für die Standardausgabe. Was rechts vom Operator `<<` steht, wird dorthin ausgegeben. Strings werden in C++ in echte Anführungszeichen eingeschlossen. Ein Zeilenwechsel kann innerhalb eines Strings durch `"\n"` dargestellt werden. Es können beliebig viele Objekte mit einer Anweisung

ausgegeben werden. Zum Beispiel kann die Anweisung

```
write ('Pi = ', 3.12459);
```

durch

```
cout << "Pi = " << 3.14159;
```

ersetzt werden.

- *Zeile 50:* `cin` steht für die Standardeingabe. Die Eingabe wird der Variablen rechts vom Operator >> übergeben. Die Wirkung ist nicht exakt die gleiche, wie in der Pascal-Version.
- *Zeile 57:* Im Umgang mit Strings unterscheiden sich Pascal und C++ wesentlich. Zur Prüfung auf Gleichheit benötigen Sie die Bibliotheksfunktion `strcmp`. Das Ergebnis ist Null genau dann, wenn beide Strings gleich sind. Die elegantere Pascal-Schreibweise werden wir später mit Hilfe des Klassenkonzepts nachbilden.

1.2 Namen

Namen (*Bezeichner*) für benutzerdefinierte Objekte (Variablen, Konstanten, Typen, Funktionen), werden fast wie in Turbo Pascal gebildet: Dort muß ein Name mit einem Buchstaben (keine Umlaute etc.) beginnen; danach dürfen Buchstaben, Ziffern und Unterstriche (_) beliebig folgen.

In C++ darf ein Name auch mit einem Unterstrich beginnen. Allerdings sind solche Namen für die Sprachimplementierung reserviert und sollen nicht vom Programmierer definiert werden. Die Anzahl der signifikanten Zeichen ist compilerabhängig. In C++ wird, im Gegensatz zu Pascal, zwischen Groß- und Kleinbuchstaben unterschieden. Die *Schlüsselwörter* von C++ sind reserviert und dürfen nicht als Namen definiert werden:

Schlüsselwörter von C++				
asm	*auto*	bool	break	case
catch	char	class	const	const_cast
continue	default	delete	do	double
dynamic_cast	else	enum	explicit	extern
false	float	for	friend	goto
if	inline	int	long	mutable
namespace	new	operator	private	protected

public	register	reinterpret_cast	return	short
signed	sizeof	static	static_cast	struct
switch	template	this	throw	true
try	typedef	typeid	typename	union
unsigned	using	virtual	void	volatile
wchar_t	while			

Die beiden kursiv gekennzeichneten Schlüsselwörter werden in diesem Buch nicht behandelt. `auto` ist ein Relikt aus früheren Versionen, `asm` weist darauf hin, daß die nächste Anweisung oder der nächste Block in Assembler formuliert ist und ist daher maschinenabhängig.

Sie sollten Ihren Objekten möglichst prägnante Namen geben. Es ist manchmal nicht ganz einfach, einen treffenden und dennoch nicht allzu langen Namen zu finden. Manchmal erfordert es ein paar Minuten Nachdenken, einen möglichst kurzen, aussagekräftigen Namen zu finden. Solche Sorgfalt zahlt sich beim späteren Bearbeiten des Programms mit Sicherheit aus. In manchen Fällen ist es übrigens durchaus angebracht, einzelne Buchstaben als Variablennamen zu verwenden, so etwa für lokale Laufvariablen oder für die Argumente abstrakter mathematischer Funktionen.

Es kann leicht zu Mißverständnissen führen, wenn Abkürzungen zusammengesetzter Wörter verschmolzen werden. Da die Syntax von C++, wie fast aller Programmiersprachen, keine Leerzeichen in einem Namen erlaubt, muß man sich hier anders behelfen. Bei Pascal- und Modula-2-Programmierern ist es üblich, die einzelnen Wörter lückenlos hintereinanderzuschreiben, aber die neuen Wörter mit einem Großbuchstaben zu beginnen (Beispiel aus Turbo Pascal: *MemAvail* ist wesentlich besser lesbar als *memavail*). Diese Konvention empfiehlt sich auch in C++. Allerdings müssen Sie wegen der Unterscheidung von Groß- und Kleinbuchstaben auf stets gleiche Schreibweise achten. Alternativ dazu können Teilwörter auch durch einen Unterstrich getrennt werden.

Für nicht englischsprachige Programmierer erhebt sich auch die Frage, ob man der Einheitlichkeit wegen selbst englische Namen oder im Interesse der Lesbarkeit muttersprachliche Namen verwenden sollte. In diesem Buch verwende ich mit Rücksicht auf viele Leser weitgehend deutsche Namen, mache aber Ausnahmen in solchen Fällen, wo bereits englische Namen weitverbreitet sind.

1.3 Kommentare

Kommentare können zum Verständnis eines Programms beitragen, wenn sie sinnvoll verwendet werden. Überflüssige Kommentare können aber sogar den Überblick über ein

Programm erschweren. Im Zweifelsfall sollten Sie zunächst versuchen, den Programmcode so klar zu gestalten, daß Kommentare überflüssig werden. Insbesondere ist es nicht der Sinn eines Kommentars, die Bedeutung eines Namens zu erklären. Das sollte der Name selbst tun. Kommentare sollten nicht wiederholen, was aus dem Programmcode unmittelbar hervorgeht. Kommentare sind insbesondere dann sinnvoll, wenn die Implementierung eines Algorithmus inhaltlich nicht unmittelbar verständlich ist.

In C++ gibt es zwei verschiedene Möglichkeiten für Kommentare. Mit /* wird der Beginn, mit */ das Ende eines Kommentars angezeigt. Hiermit können größere Kommentarblöcke eingeschlossen werden. Oft stehen kurze Kommentare am Ende einer Zeile. Hierfür gibt es in C++ eine bequeme Alternative: Durch // wird der Rest der Zeile zum Kommentar erklärt. Kommentare dürfen nicht ineinander verschachtelt werden. Es ist aber erlaubt, zwischen /* und */ Zeilenendkommentare einzuschließen.

1.4 Elementare Datentypen

Die Speicherdarstellung der Standard-Datentypen ist compilerabhängig. In der folgenden Übersicht ist jeweils die Größe in Bits für Windows 3.1 / Windows 95 aufgeführt:

Standard-Datentypen	Turbo Pascal	C++	Bits
Gleitkommatypen	`real`	-	48
	`Extended`	`long double`	80
	`Double`	`double`	64
	`Single`	`float`	32
Ganzzahltypen	-	`unsigned long`	32
	`LongInt`	`long`	32
	`Word`	`unsigned`	16/32
	`integer`	`int`	16/32
	`Word`	`unsigned short`	16
	`integer`	`short`	16
	`Byte`	`unsigned char`	8
	`ShortInt`	`char`	8

Den Datentyp `boolean` gibt es in C++ nicht. Stattdessen gilt die Konvention, daß die Zahl Null für den Wahrheitswert `false` steht, alle übrigen Zahlen für den Wahrheitswert `true`.

In den MS-DOS-Implementierungen stimmt der Datentyp `short` mit `int` überein und ist daher überflüssig.

1.5 Konstanten

Die folgende Gegenüberstellung gibt Ihnen einen Überblick über die Darstellung von Konstanten in C++:

	Turbo Pascal	**C++**
Dezimal	`122`	`122`
Sedezimal (Hexadezimal)	`$7A`	`0x7A`
Oktal	`---`	`0172`
Gleitkommazahl	`3.14`	`3.14`
	`1.4E-10`	`1.4E-10`
Zeichen	`'z'`	`'z'`
Strings (Zeichenketten)	`'Beispiel'`	`"Beispiel"`

Wie Sie sehen, unterscheidet sich die Darstellung nur bei Sedezimalzahlen und bei Strings. *Sedezimale Konstanten* werden mit den 16 Ziffern `0123456789ABCDEF` gebildet und mit `0x` eingeleitet. Dabei können die Buchstaben `A..F` und `x` (und ebenso das `E` für den Dezimal-Exponenten in Gleitkommakonstanten) wahlweise groß oder klein geschrieben werden.

Oktale Konstanten beginnen mit einer Null und dürfen nur die Oktalziffern `0` bis `7` enthalten. Auf Byte-orientierten Rechnern, wie dem IBM-PC, sind Sedezimal-Konstanten meist sinnvoller. Sie sollten daran denken, daß Sie niemals Dezimalzahlen mit führenden Nullen auffüllen, weil sie sonst als Oktalzahlen interpretiert werden!

Zahlkonstanten gehören dem kleinstmöglichen Datentyp an. So ist etwa `17` vom Typ `int`, aber `100000` vom Typ `long`. Durch den nachgestellten Buchstaben `l` oder `L` (`l` kann leicht mit `1` verwechselt werden!) können auch kleine Zahlen als `long` definiert werden. Ebenso kann durch den nachgestellten Buchstaben `u` oder `U` der Typ `unsigned` erzwungen werden.

Auch die Kombination `UL` ist möglich:

```
17       int
17L      long
40000    long (zu groß für int)
40000U   unsigned int
100000U  unsigned long (zu groß für unsigned int)
```

Nicht druckbare Zeichen-Konstanten können auch mit Hilfe ihres ASCII-Codes dargestellt werden, und zwar wahlweise durch die Zeichenfolge `"\x"`, gefolgt von bis zu drei Sedezimalziffern oder durch das Zeichen `'\'`, gefolgt von bis zu drei Oktalziffern. So können Sie zum Beispiel das Escape-Zeichen (ASCII 27) wahlweise durch `'\x1B'`, `'\x01B'`, `'\33'` oder `'\033'` darstellen.

Für einige oft benötigte *Steuerzeichen* gibt es spezielle Standarddarstellungen:

ASCII	**Kurzzeichen**	**Bezeichnung**	**Darstellung**
7	BEL	Alarm (Warnton)	`'\a'`
8	BS	Rückschritt (Backspace))	`'\b'`
9	HT	Horizontal-Tabulator	`'\t'`
10	LF	Zeilenvorschub (Line Feed)	`'\n'`
11	VT	Vertikal-Tabulator	`'\v'`
12	FF	Seitenvorschub (Form Feed)	`'\f'`
13	CR	Wagenrücklauf (Carriage Return)	`'\r'`
34	"	Anführungszeichen	`'\"'`
92	\	Backslash	`'\\'`
96	'	Apostroph ("Hochkomma")	`'\''`

In String-Konstanten können nicht druckbare Zeichen ebenso wie in Zeichenkonstanten verwendet werden. Der String

```
"Erste Zeile\nZweite Zeile"
```

enthält ein Zeilenvorschub-Zeichen.

Wenn Sie innerhalb einer Stringkonstante nichtdruckbare Zeichen in der allgemeinen Form verwenden, ist Vorsicht angebracht. Wollen Sie etwa auf einem Standard-Nadeldrucker

die Kursivschrift einschalten, so müssen Sie die Sequenz `[Esc]-'4'` senden. Sie können also erst das Zeichen `'\33'` und dann das Zeichen `'4'` senden. Wenn Sie beide Zeichen in einem String zusammenfassen, ergibt sich `"\334"`. Dies wird aber als String mit nur einem Zeichen (`'\334'`) mit dem Oktalcode `0334` oder dem Dezimalcode `220` aufgefaßt, denn der Oktalcode darf bis zu drei Stellen besitzen! Sie können dieses Problem vermeiden, indem Sie den Oktal- oder Sedezimalcode mit führenden Nullen auf drei Stellen auffüllen: `"\0334"`.

1.6 Ausdrücke und Operatoren

Wie in Pascal werden *Ausdrücke* auch in C++ aus elementaren Einheiten durch beliebige Anwendung von Operatoren gebildet. Die Unterschiede liegen in der Abgrenzung dieser Operatoren. In Pascal kann mit dem Zuweisungsoperator kein Ausdruck gebildet werden, sondern die Zuweisung ist eine spezielle Anweisung. In C++ dagegen nimmt der Zuweisungsoperator nicht diese Sonderstellung ein. Das bedeutet: Auch die Zuweisung ist in C++ ein Ausdruck und hat demnach einen Wert. Der Wert einer Zuweisung ist der Wert des zugewiesenen Ausdrucks.

Zunächst gibt es alle von Pascal her bekannten Operatoren auch in C++, allerdings teilweise in recht exzentrischer Schreibweise. Wenn Ihnen die Symbole für einige Operatoren in C++ überhaupt nicht gefallen, können Sie für eine Eingewöhnungsphase bei den Pascal-Bezeichnungen bleiben, wenn Sie die folgenden Zeilen (mit Präprozessor-Definitionen) am Anfang Ihrer Programme einfügen:

```
#define MOD  %
#define DIV  /
#define NOT  !
#define AND  &&
#define OR   ||
#define XOR  ^^
```

Arithmetische Operatoren

Arithmetische Operatoren		**Pascal**	**C++**
Unär	unäres Plus	+	+
	unäres Minus	-	-
binär	Addition	+	+
	Subtraktion	-	-
	Multiplikation	*	*
	Ganzzahl-Division	div	/

	Gleitkomma-Division	/	/
	Modulo-Operator	mod	%

Logische (boolesche) Operatoren

Logische Operatoren		Pascal	C++
unär	Negation	not	!
binär	Und-Verknüpfung	and	&&
	Oder-Verknüpfung	or	||
	exklusives Oder	xor	^^

Wie schon erwähnt, gibt es in C++ keinen speziellen Typ (`boolean`) für Wahrheitswerte. Die logischen Operatoren erwarten Argumente vom Typ `int`. Hat ein Argument den Wert Null, wird es als falsch interpretiert sonst als wahr. Die logischen Operatoren geben für den Wahrheitswert `true` immer die Zahl `1` zurück.

In Turbo Pascal kann der Compiler so eingestellt werden, daß er logische Ausdrücke nur soweit wie nötig auswertet. Hat der Ausdruck links von einem Und-Operator den Wert `false`, so steht bereits fest, daß der Operator den Wert `false` zurückliefert. Deshalb braucht die rechte Seite nicht mehr untersucht zu werden. Das entsprechende gilt, wenn auf der linken Seite eines Oder-Operators ein wahrer Ausdruck steht. In C++ ist dieses optimierte Auswerten von logischen Ausdrücken in der Sprachdefinition vorgeschrieben. Sie können sich also darauf verlassen, daß etwa die Anweisung

```
if (q > 0 && p/q == 2) r++;
```

keine Division durch Null ausführt.

Die merkwürdigen Doppelsymbole für die logischen Operatoren gehören mit Sicherheit nicht zu den ästhetischen Glanzlichtern von C++. Wie Sie sogleich sehen werden, gibt es auch die entsprechenden einfachen Symbole, allerdings in anderer Bedeutung.

Operatoren zur Bitmanipulation von Ganzzahltypen

Diese Operatoren sollten nur in begründeten Fällen verwendet werden. Sie erlauben den direkten Zugriff auf einzelne Bits. Die typischen Anwendungsfälle lassen sich mit Hilfe der Bitfelder meist wesentlich klarer und problemnäher beschreiben.

Operatoren zur Bitmanipulation		**Pascal**	**C++**
unär	bitweise Negation	`not`	~
binär	bitweises Und	`and`	&
	bitweises Oder	`or`	|
	bitweises exkl. Oder	`xor`	^
	Linksverschiebung	`shl`	<<
	Rechtsverschiebung	`shr`	>>

Die bitweisen Verschiebungen werden oft zur schnelleren Multiplikation oder Division mit Zweierpotenzen verwendet. Dies ist bei modernen optimierenden Compilern meist überflüssig und sollte daher möglichst vermieden werden. Das Rechtsschieben von vorzeichenbehafteten Zahlen ist übrigens nicht portabel. Manche Compiler schieben Nullen nach, andere schieben das Vorzeichenbit nach (arithmetische Verschiebung), damit auch negative Zahlen durch Zweierpotenzen dividiert werden können.

In Pascal werden für die logischen Operatoren und für die Bitoperatoren die gleichen Operanden verwendet. Dieses sogenannte *Überladen von Operatoren* ist in Pascal möglich, da sich dort Zahlen von Wahrheitswerten im Typ unterscheiden.

Sie können Ihr Verständnis der Operatoren an den folgenden Beispielen überprüfen:

Pascal		**C**	
Ausdruck	Wert	Ausdruck	Wert
`(3>2) and (3<4)`	`true`	`3>2 && 3<4`	`1`
		`3>2 & 3<4`	`1`
`5 and 6`	4	`5 & 6`	4
		`5 && 6`	`1`
`(3>2) and 4`	Syntaxfehler	`3>2 && 4`	`1`
		`3>2 & 4`	`0`

In Pascal werden einige der Operatoren auch noch für Mengenoperationen und String-Verkettung benutzt. Das Überladen von Operatoren steht in C++ generell dem

Programmierer als Sprachmittel zur Verfügung. Wir werden später sehen, daß sich so das Mengenkonzept von Pascal identisch in C++ realisieren läßt, obwohl es nicht zur Sprache gehört.

Vergleichsoperatoren

Hier genügt eine tabellarische Gegenüberstellung

Vergleichsoperatoren	**Pascal**	**C++**
Gleichheit	=	==
Ungleichheit	<>	!=
kleiner	<	<
kleiner oder gleich	<=	<=
größer	>	>
größer oder gleich	>=	>=

Nur die Gleichheits- und die Ungleichheitsrelation werden anders als in Pascal geschrieben. Schreiben Sie in C++ versehentlich <> anstelle von `!=`, weist Sie der Compiler darauf hin. Wenn Sie dagegen die Gleichheitsrelation (==) in der von Pascal gewohnten Weise (=) schreiben, kann das unangenehme Folgen haben, wie Sie sogleich sehen werden.

Zuweisungoperatoren

Die Zuweisung wird in C++ mit dem einfachen Gleichheitszeichen (ohne vorgestellten Doppelpunkt!) dargestellt. Wie Sie bereits zu Beginn dieses Abschnitts erfahren haben, sind in C++ auch Zuweisungen Ausdrücke mit einem Wert, nämlich dem des zugewiesenen Ausdrucks. Damit sind auch Zuweisungsketten möglich: `a = b = 1;` ist gleichbedeutend mit `a = (b = 1);`. Der Wert von `(b = 1)` ist `1` und wird `a` zugewiesen.

Beim Umstieg von Pascal nach C++ tut sich hier eine der berüchtigtsten Fallgruben auf:

```
a := 0;
if a = 1 then write ('ja')
else write ('nein')
```

```
a = 0;
if (a = 1) printf ("ja");
else printf ("nein");
```

Das Pascal-Programm sagt "nein", C++ entscheidet sich für "ja". Was ist der Grund? Der Ausdruck nach dem `if` ist im Pascal-Programm eine Gleichheitsrelation mit dem Ergebnis `false`, in C++ eine Zuweisung mit dem Ergebnis `1`, was von Null verschieden und damit wahr ist! Würden Sie in Pascal den gleichen Fehler machen, nämlich versehentlich `a := 1` anstelle `a = 1` schreiben, würde der Compiler das nicht durchgehen lassen, weil nach dem `if` nur ein Ausdruck, aber keine Anweisung stehen darf.

Auf der linken Seite einer Zuweisung muß, ebenso wie in Pascal, ein Objekt stehen, dem eine bestimmte Speicheradresse zugeordnet ist. In der C-Terminologie spricht man von einem *L-Wert* (*L-value*, L für linke Seite). Auf der rechten Seite darf ein beliebiger Ausdruck stehen. Dieser Ausdruck muß vom gleichen Typ wie der L-Wert sein oder in den L-Wert umgewandelt werden können. Die genauen Regeln für solche Typkonvertierungen erfahren Sie in einem späteren Abschnitt.

Neben der gewöhnlichen Zuweisung gibt es noch eine ganze Reihe von Kombinationen von Zuweisungen mit arithmetischen Operationen. Anstelle von

```
a = a + b;
```

können Sie auch die Kurzschreibweise

```
a += b;
```

verwenden. Dies ist besonders bei komplizierten Objekten von Vorteil:

```
langeVariable[2*i+j] = langeVariable[2*i+j] + 2;
```

ist mühsamer zu schreiben und nicht so leicht lesbar wie

```
langeVariable[2*i+j] += 2;
```

Zudem macht diese Schreibweise dem Compiler klar, daß er die Adresse der Variablen nicht zweimal berechnen muß.

In entsprechender Bedeutung gibt es die Operatoren

```
+=     -=     *=     /=     %=     >>=     <<=     &=     ^=     |=
```

Besonders oft kommt eine Erhöhung oder Verminderung einer Variablen um eins vor. Dafür hat Turbo Pascal von Modula-2 die Prozeduren `INC` und `DEC` übernommen. C++ bietet hierfür die unären Operatoren ++ und -- an. Statt `a += 1` können Sie auch a++ oder ++a schreiben, `a -= 1` können Sie durch a-- oder --a ersetzen. Zwischen der Präfix- und der Postfix-Schreibweise gibt es einen feinen Unterschied, der nur dann von Bedeutung ist, wenn der Wert eines solchen Ausdrucks ausgewertet werden soll: Der Wert eines Präfixausdrucks ist der neue Wert der Variablen, der des Postfix-Ausdrucks der alte Wert. Jetzt können Sie sich auch die Bedeutung des Namens C++ erklären!

Das Programmstück

```
a = 2;
while (a--) printf ("A");
a = 2;
while (--a) printf ("B");
```

schreibt 2 mal "A", aber nur einmal "B".

Der Operator sizeof

Mit diesem Operator, den es ähnlich auch in Turbo Pascal gibt, kann die Größe eines Datentyps oder eines Ausdrucks in Bytes ermittelt werden. Er wird formal wie eine Funktion verwendet. Beispiele:

```
int i, j;
i = sizeof (int);
j = sizeof (i);
```

Der Bedingungsoperator

Der Bedingungsoperator hat eine etwas ungewöhnliche Syntax. Er ist der einzige dreistellige Operator in C++. Er erlaubt es, den Wert eines Ausdrucks von einer Bedingung abhängig zu machen, ohne eine `if`-Anweisung zu verwenden. Die Syntax lautet:

logischerAusdruck ? **Ausdruck1** : **Ausdruck2**

Der Wert eines solchen bedingten Ausdrucks ist der Wert von **Ausdruck1** bzw. **Ausdruck2**, je nachdem, ob der logische Ausdruck wahr oder falsch ist. Es folgen einige Beispiele:

ohne Bedingungsoperator	mit Bedingungsoperator
`if (a < b) A[i][j] = x;` `else A[i][j] = y;`	`A[i][j] = (a < b) ? x : y;`
`if (x < y) return x;` `else return y;`	`return (x < y) ? x : y;`

Der Komma-Operator

Dieser Operator wird hier nur der Vollständigkeit halber erwähnt. Er ist ein hervorragendes Mittel, Programme unlesbar zu machen. Sie sollten ihn nur in Ihren passiven Sprachschatz aufnehmen.

Zwei Ausdrücke können durch ein Komma dazwischen zu einem Ausdruck verschmolzen werden:

Ausdruck1 , **Ausdruck2**

Bei der Auswertung werden der Reihe nach beide Ausdrücke ausgewertet. Der Wert des Gesamtausdrucks ist der Wert des zweiten Ausdrucks. Einige abschreckende Beispiele zeigen die Möglichkeiten, mit dem Komma-Operator Verwirrung zu schaffen:

```
if (x > 0) i++, j++; k++    // k wird immer erhöht, i und j, falls x > 0
x = sin (3,14159)           // Fehler: sin erwartet nur ein Argument
x = sin ((3,14159));        // sin (14159) wird berechnet
x = sin ((Pi = 3.14159, Pi/2)); // sin (Pi/2) wird berechnet
```

Die Vorrangregeln

In Pascal müssen Sie ziemlich viele Klammern setzen, weil es nur vier Vorrangstufen gibt:

Am stärksten bindet der `not`-Operator, danach kommen die multiplikativen Operatoren (`*`, `/ div`, `mod`, `and`), dann die additiven Operatoren (`+`, `-`, `or`) und schließlich, mit schwächster Bindung, die Vergleichsoperatoren (=, <, >, <>, >=, <=, `in`).

C++ bietet eine 16-stufige Operatorenhierarchie. Deshalb ist die einfachste Faustregel: Wenn Sie die Vorrangregeln nicht ganz sicher beherrschen, sollten Sie lieber einige Klammern zuviel als zuwenig verwenden.

Eine vollständige Tabelle aller Operatoren, auch der Ihnen bisher unbekannten, finden Sie im Anhang. Hier sollen einige Hinweise genügen:

Sehr oft kommen logische Verknüpfungen von Vergleichsrelationen vor. In Pascal müssen die Vergleiche geklammert werden, in C++ ist das überflüssig:

```
if (a > 0) and (b > 0) then ...        if (a > 0 && b > 0) ...
```

Einige Operatoren verhalten sich anders als Sie es vielleicht erwarten. So binden die Schiebe-Operatoren schwächer als die Addition und Subtraktion. Die Ausdrücke `a * 16 + 1` und `a << 4 + 1` sind nicht äquivalent! Sie werden nämlich so interpretiert: `(a * 16) + 1` und `a << (4 + 1)`.

Unäre (einstellige) Operatoren und Zuweisungsoperatoren sind *rechtsassoziativ*. Das heißt: Wenn rechts und links von einem Ausdruck ein solcher Operator steht, wird zunächst der rechte Operator angewendet. So steht etwa `*p++` für `*(p++)` oder `*a[i]` für `*(a[i])` oder `x = y = z` für `x = (y = z)`.

Alle übrigen binären Operatoren sind *linksassoziativ*. `x   y   z` bedeutet also `(x   y)   z`.

1.7 Anweisungen

Im letzten Abschnitt haben Sie bereits erfahren, daß Zuweisungen in C++ keine Anweisungen, sondern Ausdrücke sind. Jeder Ausdruck wird aber zu einer Anweisung, wenn man ein Semikolon anhängt. Es folgen ein paar Beispiele:

Ausdrücke	**Anweisungen**
`a = b + 1`	`a = b + 1;`
`a++`	`a++;`
`a + 3`	`a + 3;`

Nicht jede Anweisung ist wirklich sinnvoll. So berechnet die dritte Beipiel-Anweisung die Summe `a + 3`, ohne daß dies irgendwelche Konsequenzen hätte. Der Wert von a bleibt unverändert.

Beachten Sie auch, daß in C++ das abschließende Semikolon Bestandteil der Anweisung ist, während in Pascal das Semikolon nicht zu einer Anweisung gehört, sondern als Trenner in Verbundanweisungen dient. Dieser feine Unterschied ist wichtig!

Wir gehen nun im einzelnen auf die verschiedenen Anweisungsformen ein. Dabei läßt es sich nicht ganz vermeiden, daß wir in einigen Beispielen schon auf später erklärte Sprachmittel zurückgreifen.

Die leere Anweisung

Die *leere Anweisung* besteht lediglich aus einem Semikolon. Sie wird zum Beispiel benötigt, wenn mit einer while-Anweisung untätig auf das Eintreffen eines Ereignisses, wie etwa eines Tastendrucks gewartet wird:

```
while (keineTasteGedrueckt);
```

Diese Anweisung entlockt dem Zortech-Compiler eine Warnung. Sie können diese Warnung vermeiden, indem Sie das Schlüsselwort void für die leere Anweisung einsetzen:

```
while (keineTasteGedrueckt) void; // In Borland C++ verboten!
```

Die Verbundanweisung

Wie in Pascal verlangt die Syntax an manchen Stellen genau eine Anweisung. Deshalb muß es möglich sein, aus mehreren Anweisungen formal eine Anweisung zu machen. Hierzu werden in C++ anstelle der Pascal-Schlüsselwörter begin und end die geschweiften Klammern verwendet. In C++ wird durch eine Verbundanweisung ein lokaler Block eingeführt. Variablen, die hier deklariert werden, sind außerhalb der Verbundanweisung nicht sichtbar, und ihr Speicherplatz wird nach Beendigung der Verbundanweisung wieder freigegeben.

Die if-Anweisung

if **Ausdruck** then **Anweisung**
[else **Anweisung]**

if (**Ausdruck**) **Anweisung**
[else **Anweisung]**

Die if-Anweisung ist in C++ strukturgleich mit Pascal: Sie könnten die gewohnte Pascal-Schreibweise beibehalten und die Übersetzung dem Präprozessor überlassen. Dazu müßten Sie folgende Definitionen in Ihr Programm einfügen:

```
#define IF      if (
#define THEN    )
#define ELSE    else
```

Genaueres über den Präprozessor erfahren Sie im Abschnitt 2.1. Ich empfehle Ihnen nicht, diese Präprozessor-Definitionen wirklich zu verwenden. Sie sollen Ihnen lediglich die Umsetzung von Pascal nach C++ klarmachen. Wie Sie sehen, wird in C++ die Bedingung in runde Klammern gesetzt, dafür entfällt das Schlüsselwort then.

Die folgenden beiden Zeilen

```
IF a < 0 THEN b := 0;
IF a < 0 THEN b := 0; ELSE b := 1;
```

würden expandiert zu den korrekten C++-Anweisungen

```
if (a < 0) b = 0;
if (a < 0) b = 0; else b := 1;
```

Beachten Sie das Semikolon vor dem `else`! Da in C++ jede Anweisung mit einem Semikolon endet, muß auch hier eines stehen. In Pascal wäre dies ein Fehler. Gemeinsam ist beiden Sprachen, daß nach dem `then` und dem `else` jeweils nur eine Anweisung stehen darf. Mehrere Anweisungen müssen zu einer Verbundanweisung gemacht werden.

```
if x < y then begin
  z := x;
  x := y;
  y := z;
end
```

```
if (x < y) {
  z = x;
  x = y;
  y = z;
}
```

In der Sprachdefinition von C++ wird (wie in Pascal) festgelegt, daß in syntaktisch mehrdeutigen Anweisungen wie der folgenden

`if (`**Ausdruck**`) if (`**Ausdruck**`)` **Anweisung** `else` **Anweisung**

jedes `else` dem letztmöglichen `if` zugeordnet wird.

Da es in C++ keinen Datentyp `boolean` gibt, darf für den Bedingungsausdruck ein beliebiger arithmetischer Ausdruck oder ein Ausdruck eines Zeigertyps stehen. Die Bedingung ist erfüllt, wenn der Ausdruck von Null verschieden ist.

Die while-Anweisung

`while` **Ausdruck** `do` **Anweisung**

`while (` **Ausdruck** `)` **Anweisung**

Auch die `while`-Anweisung ist strukturgleich mit Pascal. Sie könnte mit folgenden Präprozessor-Anweisungen übersetzt werden:

```
#define WHILE  while (
#define DO     )
```

Wie bei der `if`-Anweisung wird in C++ auch hier die Bedingung in runde Klammern gesetzt. Dafür entfällt das Schlüsselwort `do`.

Die do-Anweisung

Die `repeat`-Anweisung aus Pascal wird in C++ durch die `do`-Anweisung ersetzt.

`repeat` **Anweisungsfolge**
`until` **Ausdruck**

`do` **Anweisung**
`while (` **Ausdruck** `)`

Beachten Sie bitte folgende Unterschiede:

- `repeat` wird durch `do` ersetzt, `until` durch `while`.
- Statt einer Abbruchbedingung in Pascal wird in C++ eine Wiederholungsbedingung angegeben, also das logische Gegenteil. Die Bedingung wird, wie in der `if`- und

while-Anweisung auch, in runde Klammern gesetzt.

- Die Syntax der repeat-Anweisung fällt in Pascal etwas aus dem Rahmen. Sie ist die einzige strukturierte Anweisung, in die nicht nur eine, sondern beliebig viele Unteranweisungen eingesetzt werden dürfen, ohne sie zu einer Verbundanweisung zu klammern. Die Klammerung übernimmt hier das abschließende Schlüsselwort until. Obwohl die entsprechende Anweisung in C++ formal gleichartig aufgebaut ist, darf trotzdem nur genau eine Anweisung eingesetzt werden.
-

```
repeat                                  do {
  i := i * i;                             i = i * i;
  j := j + i                              j = j + i;}
until i > n                             while (i <= n);
```

Das Schlüsselwort while wird in C++ also für zwei verschiedene Zwecke eingesetzt: Für die Anfangsabfrage der while-Anweisung und für die Endabfrage der do-Anweisung.

Das Schlüsselwort do steht in C++ einzig für die do-Schleife. In Pascal wird es für ganz andere Zwecke verwendet (nach while, for und with).

Die for-Anweisung

Die for-Anweisung in Pascal ist nicht gerade üppig ausgestattet:

for **Variable** := **Ausdruck** to **Ausdruck** do **Anweisung**

oder

for **Variable** := **Ausdruck** downto **Ausdruck** do **Anweisung**

Zumindest eine von ±1 abweichende Schrittweite wäre wünschenswert. Was hier allerdings die entsprechende Konstruktion in C++ bietet, ist so mächtig, daß wir uns zunächst mit der direkten Umsetzung einer for-Schleife aus Pascal begnügen wollen:

```
for i := 1 to 10 do                     for (i = 1; i <= 10; i = i + 1)
  s := s + i;                             s = s + i;
for i := 10 downto 1 do                 for (i = 10; i >= 1; i = i - 1)
  s := s + i;                             s = s + i;
```

Nach diesem Schema können alle for-Anweisungen aus Pascal übertragen werden. Doch die merkwürdig anmutende Ausdrucksweise in C++ läßt erahnen, daß die Möglichkeiten damit bei weitem nicht ausgeschöpft sind. Die for-Anweisung hat in C++ die allgemeine Form

for (**Anweisung [Ausdruck]** ; **[Ausdruck]**) **Anweisung**

Da in C++ jede Anweisung mit einem Semikolon endet, stehen zwischen den runden Klammern also zwei Semikola!

Eine Anweisung der Form

for (**Anfangsanweisung Fortsetzungsbedingung** ; **Inkrementierung**)

Schleifenanweisung

hat die gleiche Bedeutung wie das Programmstück

```
Anfangsanweisung
while ( Fortsetzungsbedingung ) {
   Schleifenanweisung
   Inkrementierung
}
```

Damit läßt sich unter anderem jede while-Schleife darstellen, indem man die Anfangsanweisung und die Inkrementierung wegläßt:

```
while ( Bedingung ) Anweisung
```

könnte (aber sollte keinesfalls) ersetzt werden durch

```
for ( ; Bedingung ; ) Anweisung
```

Eine fehlende Fortsetzungsbedingung wird als wahr interpretiert. Durch eine for-Anweisung der Form

```
for ( ; ; ) Anweisung
```

läßt sich also eine Endlosschleife darstellen.

Die switch-Anweisung

Mit der case-Anweisung in Pascal kann die folgende Anweisung vom Wert einer ordinalen Selektor-Variablen abhängig gemacht werden. Der gleiche Mechanismus steht auch in C++ zur Verfügung.

```
case Monat of
  1: Tage := 31;
  2: Tage := 28;
  3: Tage := 31;
  4: Tage := 30;


  5: Tage := 31;
  6: Tage := 30;
  7: Tage := 31;
  8: Tage := 31;
  9: Tage := 30;
 10: Tage := 31;
 11: Tage := 30;
 12: Tage := 31;
end;
```

```
switch (Monat) {
  case  1: Tage = 31; break;
  case  2: Tage = 28; break;
  case  3: Tage = 31; break;
  case  4: Tage = 30; break;

  case  5: Tage = 31; break;
  case  6: Tage = 30; break;
  case  7: Tage = 31; break;
  case  8: Tage = 31; break;
  case  9: Tage = 30; break;
  case 10: Tage = 31; break;
  case 11: Tage = 30; break;
  case 12: Tage = 31; break;
}
```

Das Pascal-Schlüsselwort case wird in C++ durch switch ersetzt. Der Selektorausdruck wird in C++ in runde Klammern gesetzt, dafür entfällt das Wort of. Vor jeden einzelnen Fall muß das Wort case gesetzt werden. Der wichtigste Unterschied liegt in der Art der

Abarbeitung. Pascal führt nur die Anweisung nach der zutreffenden Konstanten aus, C++ dagegen alle Anweisungen ab der zutreffenden Konstante. Wird dies nicht gewünscht (das ist der Normalfall!), muß die Abarbeitung der weiteren Anweisungen durch das Schlüsselwort `break` verhindert werden. Nur durch diese merkwürdige Konvention ist es in C++ möglich, mehreren Konstanten die gleiche Anweisung zuzuordnen:

```
case Monat of
  1,3,5,7,8,10,12: Tage := 31;
  2              : Tage := 28;
  4,6,9,11       : Tage := 30;
else Tage := 0
end;
```

```
switch (Monat) {
  case  1:
  case  3:
  case  5:
  case  7:
  case  8:
  case 10:
  case 12: Tage = 31; break;
  case  2: Tage = 28; break;
  case  4:
  case  6:
  case  9:
  case 11: Tage = 30; break;
  default: Tage = 0;
}
```

Hier bietet Pascal die elegantere Form. In diesem Beispiel sehen Sie auch, wie Werte behandelt werden können, zu denen es keine `case`-Konstante gibt. Dafür dient das Schlüsselwort `default` (Das entsprechende `else` im Pascal-Beispiel ist eine spezielle Spracherweiterung von Turbo Pascal!).

Die `case`-Anweisung in Pascal und die `switch`-Anweisung in C++ haben folgende Gemeinsamkeiten: Die Reihenfolge der Konstanten ist beliebig, es darf auch Lücken zwischen den Konstanten geben. Alle angegebenen Konstanten müssen verschieden sein.

Sie sollten sich klarmachen, daß das Resultat einer `case`/`switch`-Anweisung extrem von der Implementierung des Compilers abhängt. Während etwa `if`- oder `while`-Anweisungen in stets gleichbleibender Weise in Maschinensprache umgesetzt werden, hängt die optimale Umsetzung einer `switch`-Anweisung vom individuellen Fall ab. Gibt es nicht zu viele Lücken zwischen den einzelnen Konstanten, bietet sich eine Sprungtabelle an, die durch den Wert der Selektor-Variablen indiziert wird. Wenige Konstanten, die aber weit auseinanderliegen, sollten in eine Abfrage der Form `if` ... `else if` ... umgesetzt werden. Bei einer größeren Anzahl von weit gestreuten Konstanten bieten sich dagegen Hash-Verfahren oder binäre Suche an. Mit der optimalen Umsetzung in allen Fällen dürften die meisten Compiler wohl überfordert sein. Eine Anweisung wie etwa

```
switch (Preis) {
  case   99: Anweisung1; break;
  case  199: Anweisung2; break;
  case  299: Anweisung3; break;
  ...
  case 9999: Anweisung99; break;
  default  : Anweisung0; break;
}
```

läuft unter Borland C++ um ein vielfaches langsamer als die äquivalente Form:

```
if (Preis % 100 == 99)
  switch (Preis / 100) {
    case  0: Anweisung1; break;
    case  1: Anweisung2; break;
    case  2: Anweisung3; break;
    ...
    case 99: Anweisung99; break;
    default: Anweisung0; break;
  }
else  Anweisung0;
```

Übungsaufgabe 1:

Machen Sie sich die Wirkung des folgenden Programmstücks klar und ersetzen Sie es durch eine einfache Anweisung:

```
if (a > 0 && a <= 4) {
  b = 1;
  switch (a) {
    case 4: b++;
    case 3: b++;
    case 2: b++;
  }
}
else b = a;
```

Sprunganweisungen

Bei der `while`-Anweisung wird die Fortsetzungsbedingung am Anfang abgefragt, bei der `do`-Schleife am Ende. Im Prinzip würde eine der beiden Konstuktionen ausreichen. So läßt sich etwa die allgemeine `while`-Anweisung

```
while (Bedingung)
do  Anweisung
```

ersetzen durch

```
if (Bedingung)
  do  Anweisung
  while (Bedingung)
```

Trotzdem empfinden Sie es sicher als angenehm, daß Sie in beiden Fällen die maßgeschneiderte Schleifensteuerung verwenden können. Nun gibt es aber auch Schleifen, bei denen die Fortsetzungsbedingung logisch weder am Anfang noch am Ende sondern irgendwo in der Mitte des Anweisungsblocks steht. Im folgenden Beispiel sollen solange ganze Zahlen eingegeben und deren Kehrwerte aufsummiert werden, bis eine Null eingegeben wird. Mit unseren bisherigen Mitteln könnten wir das etwa so formulieren:

```
#include <iostream.h>

main () {
  int   n;
  float s       = 0.0;
  int   Abbruch = 0;
```

```
  while (! Abbruch) {
    cin >> n;
    if (n == 0) Abbruch = 1;
    else s += 1.0 / n;
  }
  cout << "Summe = " << s;
}
```

Hier muß eine künstlich anmutende logische Variable Abbruch eingeführt und der letzte Teil des Anweisungsblocks bedingt ausgeführt werden. Das Beispiel wird durchsichtiger, wenn auch hier eine maßgeschneiderte Konstruktion verwendet wird, nämlich eine Endlosschleife mit der Abbruchbedingung in der Mitte:

```
#include <iostream.h>

main () {
  int   n;
  float s = 0.0;

  for (;;) {                 // Endlosschleife
    cin >> n;
    if (n == 0) break;       // bedingter Schleifenabbruch
    s += 1.0 / n;
  }
  cout << "Summe = " << s;
}
```

Die break-Anweisung beendet vorzeitig eine Schleife (while, do oder for) oder, wie Sie bereits wissen, eine switch-Anweisung. Bei mehreren geschachtelten Schleifen wird jeweils nur die aktuelle Schleife verlassen.

Eine weitere, allerdings seltener benötigte Sprunganweisung ist die continue-Anweisung. Sie bewirkt einen Sprung hinter die letzte Anweisung der Schleife. Das bedeutet: Die Schleife wird mit der nächsten Bedingungsabfrage fortgesetzt, bei der for-Schleife wird vorher noch die Inkrementierungsanweisung ausgeführt.

Wie in Pascal gibt es auch in C++ aus historischen Gründen eine allgemeine Sprunganweisung (goto), deren Gebrauch allerdings nur in seltenen Fällen zu rechtfertigen ist (z.B. Sprung aus verschachtelten Schleifen).

Übungsaufgabe 2:

Was gibt folgendes Programm aus:

```
#include <iostream.h>
main () {
  int n=0;
  for (int i=0; i<30; i++) {
    if (i > 20) break;
    n += 100;
    if (i < 10) continue;
    n++;
  }
  cout << n;
}
```

1.8 Funktionen

In Pascal werden gewöhnliche Prozeduren mit dem Schlüsselwort `procedure` eingeleitet, Funktionsprozeduren mit dem Schlüsselwort `function`. Im Prinzip wäre diese Unterscheidung nicht nötig, denn Funktionsprozeduren kann man ja schon an der Angabe des Ergebnistyps erkennen. Deshalb verzichtet Modula-2 auf das Schlüsselwort `FUNCTION` und verwendet das Schlüsselwort `PROCEDURE` für beide Arten von Prozeduren. In C++ dagegen spricht man nur von Funktionen. Eine Prozedur wird also in C++ als eine Funktion mit leerem Wertebereich (Pseudo-Datentyp: `void`) angesehen. Die Syntax der Prozedurdefinitionen ist so entworfen, daß kein spezielles Schlüsselwort benötigt wird.

Das folgende Beispiel zeigt in beiden Sprachen eine Funktion, die das Maximum dreier Zahlen ermittelt:

```
function max3 (a,b,c: real): real;
  var max: real;

  begin
    if a > b then max := a
    else max := b;
    if c > max then max := c;
    max3 := max;
  end;
```

```
float max3 (float a, float b,
            float c) {
  float max;

  if (a > b) max = a;
  else max = b;
  if (c > max) max = c;
  return max;
}
```

Beachten Sie bitte folgende Unterschiede in C++:

- Im *Funktionskopf* steht der Typ des Funktionswertes am Anfang, dafür entfällt das Schlüsselwort `function`.
- In der *Parameterliste* muß zu jedem einzelnen Parameter der Datentyp angegeben werden, und zwar vor dem Namen.
- Bei einer *leeren Parameterliste* dürfen die runden Klammern nicht weggelassen werden. Beispiel: `int ohneParameter ()`. Wenn Sie die Klammern vergessen, ist das gefährlich, weil dann statt des Funktionswerts die Adresse der Funktion gemeint ist, was nicht immer einen Syntaxfehler nach sich zieht. Mehr darüber erfahren Sie in einem eigenen Abschnitt.
- *Lokale Variablen* können an beliebiger Stelle innerhalb des Funktionsblocks deklariert werden.
- Der *Funktionswert* wird mit der `return`-Anweisung zurückgegeben. Damit wird gleichzeitig die Ausführung der Funktion abgebrochen.
- *Prozeduren* werden wie Funktionen mit dem Ergebnistyp `void` behandelt.

In Pascal ist eine beliebige *Verschachtelung von Prozeduren und Funktionen* erlaubt: eine Prozedur kann lokal zu einer anderen Prozedur sein. Damit ist es möglich, daß für eine Unterprozedur die lokalen Variablen der aufrufenden Prozedur global sind. Die wenigen sinnvollen Anwendungsfälle hierfür können meist mit Hilfe des Klassenkonzepts von C++ besser dargestellt werden. C++ verzichtet auf ein hierarchisches Prozedurkonzept.. Es wird nicht einmal ein formaler Unterschied zwischen dem Hauptprogramm und den Prozeduren gemacht: Das Hauptprogramm ist in C++ einfach eine spezielle Funktion mit

dem reservierten Namen `main`.

Die *Reihenfolge der Prozeduren* ist in Pascal mit Rücksicht auf Einschritt-Compiler eingeschränkt: eine aufgerufene Funktion muß im Programmtext vor der aufrufenden Funktion erscheinen. Wenn dies nicht möglich ist (wie etwa bei zwei sich wechselseitig aufrufenden Prozeduren), muß vor dem Aufruf eine `forward`-Deklaration stehen. Dies ist in C++ nicht anders: Wenn eine Funktion aufgerufen wird, muß sie in der gleichen Datei vorher deklariert sein. Wenn Sie sich von den Anordnungszwängen frei machen wollen, können Sie einfach Deklarationen für alle verwendeten Funktionen (im C-Jargon: *Prototypen*) an den Anfang des Programms setzen.

In Pascal dürfen nur einfache Typen als Funktionswert zurückgegeben werden. Dies ist ein ernsthafter Verstoß gegen das *Typvollständigkeitsprinzip*. So ist es mit der naheliegenden Definition

```
type komplex = record x, y: real end;
```

nicht möglich, Funktionen wie etwa

```
function komplexeSumme (z1, z2: komplex): komplex;
```

zu schreiben. Dieses Manko umgeht man in der Regel durch eine Prozedur mit einem zusätzlichen Referenzparameter für den Funktionswert.

In C++ gibt es diese Einschränkung nicht: Jeder Typ kann als Funktionswert verwendet werden.

Referenzen

Betrachten wir zunächst das einfache Problem, zwei Variablen mit Hilfe einer Prozedur zu vertauschen: Ein typischer Anfängerfehler wäre es, dies in Pascal so zu formulieren:

```
procedure tausche (a, b: integer); (* ohne Wirkung *)
  var h: integer;
  begin
    h := a;
    a := b;
    b := h;
  end;
```

Warum ist diese Prozedur wirkungslos? Pascal kennt zwei Arten der Parameterübergabe an Prozeduren: Im Normalfall wird der Wert des aktuellen Parameters an den formalen Parameter übergeben (*Wertparameter*). Die Prozedur kann nun den formalen Parameter verändern, ohne daß sich dies auf den aktuellen Parameter auswirkt. In unserem Beispiel sollen aber die übergebenen Variablen duch die Prozedur verändert werden. In diesem Fall muß man *Referenzparameter* (Variablenparameter) verwenden. Dann arbeitet die Prozedur nicht mit einer Kopie der formalen Variablen sondern mit diesen selbst. Dies wird in Pascal durch Voranstellen des Schlüsselworts `var` ausgedrückt:

```
procedure tausche (var a, b: integer); (* korrekte Version *)
  var h: integer;
  begin
    h := a;
    a := b;
    b := h;
  end;
```

Gäbe es keine Referenzparameter in Pascal, so könnte man das Problem auf folgende Weise angehen:

```
type IntPtr = ^integer;

procedure tausche (a, b: IntPtr);
  var h: integer;
  begin
    h := a^;
    a^ := b^;
    b^ := h;
  end;
```

Hier wurden nicht die Werte der Variablen, sondern ihre Adressen übergeben. Da es sich um Wertparameter handelt, kann die Prozedur zwar nicht diese Adressen ändern. Die Adressen sind aber der Schlüssel zum Zugriff auf die eigentlichen Zahlenwerte! In Turbo-Pascal könnte man diese Prozedur etwa so aufrufen:

```
program Test1;
type
  IntPtr = ^integer;

procedure tausche (pa, pb: IntPtr);
  var h: integer;
  begin
    h := pa^;
    pa^ := pb^;
    pb^ := h;
  end;

var a, b: integer;
begin
  a := 1;
  b := 2;
  WriteLn (a:4, b:4);
  tausche (Addr(a), Addr(b));
  WriteLn (a:4, b:4);
end.
```

Wenn Sie diese beiden Varianten vergleichen, wären Sie sicher nicht begeistert, wenn Sie von nun an auf Referenzparameter verzichten müßten. In Standard-C gibt es aber tatsächlich nur Wertparameter. Deshalb sind C-Programmierer zu einer Zeigerakrobatik verurteilt, wie wir sie gerade gesehen haben. Zum Glück wurde dieses Manko beim Übergang von C nach C++ beseitigt. Es folgen jetzt die Versionen unseres Beispiels in C und C++:

Veränderung von Parametern in C: Wertparameter und Adreßübergabe

```
void tausche (int* pa, int* pb) {
  int h;
```

Veränderung von Parametern in C++: Re

```
void tausche (int& a, int& b) {
  int h;
```

```
  h = *pa;
  *pa = *pb;
  *pb = h;
}

void main () {
  int a, b;

  a = 1;
  b = 2;
  printf ("%4%4\n", a, b);
  tausche (&a, &b);
  printf ("%4%4\n", a, b);
}
```

```
  h = a;
  a = b;
  b = h;
}

void main () {
  int a, b;

  a = 1;
  b = 2;
  printf ("%4%4\n", a, b);
  tausche (a, b);
  printf ("%4%4\n", a, b);
}
```

Sie sehen also, daß sich die Prozedur in C++ ganz wie in Pascal programmieren läßt. Um einen formalen Parameter als Referenzparameter zu kennzeichnen, stellt man das Zeichen & hinter den Typnamen. Wird das Zeichen & nicht in einer Parameterliste (und nicht in einer Variablendeklaration) verwendet, so wirkt es wie in C als Adreß-Operator. Den werden Sie aber in C++ nur selten benötigen.

Referenzparameter sollten Sie immer dann verwenden, wenn eine Prozedur die ihr übergebenen Parameter verändert. Aber auch bei unveränderlichen Parametern kann eine Referenzübergabe in manchen Fällen gerechtfertigt sein, nämlich dann, wenn eine Prozedur mit so speicheraufwendigen Parametern arbeitet, daß das Kopieren eines Parameters gegenüber der Adreßübergabe wesentlich aufwendiger wäre. Sie sollten sich aber auch über die Nachteile solcher *technischen Referenzparameter* klar sein. Damit verschenken Sie die Möglichkeit, beliebige Ausdrücke zu übergeben und automatische Typkonvertierung zu erzwingen.

C++ geht sogar noch einen Schritt weiter als Pascal und erlaubt die Übergabe von Referenzen nicht nur in Parameterlisten. Betrachten wir folgendes Programmstück:

```
if (a[10*i+b[k]] == 3 || a[10*i+b[k]] == 5)
  a[10*i+b[k]] = -a[10*i+b[k]];
```

Hier wird mehrmals das gleiche Objekt angesprochen. Fälle dieser Art bedeuten erstens hohen Schreibaufwand und zweitens muß (falls der Compiler hier nicht optimiert) mehrmals die gleiche Adresse berechnet werden. Überdies kann man nicht auf den ersten Blick erkennen, daß die Ausdrücke identisch sind. Eine Hilfsvariable kann dieses Problem teilweise, aber nicht ganz mildern:

```
int t = a[10*i+b[k]];
if (t == 3 || t == 5)
  a[10*i+b[k]] = -t;
```

Als Ziel der Zuweisung muß weiterhin das Originalobjekt angegeben werden. C++ erlaubt auch *Referenzvariablen*, die ebenso wie Referenzparameter als Synonyme für eine andere Variable stehen:

```
int& t = a[10*i+b[k]];    // t ist Referenz auf a[10*i+b[k]]
```

```
if (t == 3 || t == 5)
  t = -t;
```

Bevor eine Referenzvariable als Synonym für ein anderes Objekt verwendet werden kann, muß sie mit diesem Objekt verbunden werden. Dies geschieht bei der zwingend vorgeschriebenen Initialisierung.

Betrachten Sie bitte das Programmfragment:

```
int i = a;
i = b;
```

Hier wird zunächst die Variable i definiert, der Anfangswert a zugewiesen und anschließend mit dem Wert b überschrieben. Sie sehen, daß die Wirkung einer Initialisierung und einer separaten Zuweisung identisch ist (Bei der Behandlung von Klassen werden Sie sehen, daß es doch einen wichtigen Unterschied zwischen Initialisierung und Zuweisung gibt). Bei Referenzvariablen sieht es völlig anders aus. Hier wird durch die Initialisierung vereinbart, zu welchem Objekt die Variable synonym sein soll. Danach steht die Variable dann für ihr Synonym. Es folgt das entspechende Beispiel:

```
int& i = a;     // i ist Referenz auf a
i = b;          // gleichbedeutend mit a = b
```

Besonders klar wird sich der Nutzen von Referenzen beim Überladen von Operatoren erweisen. So werden wir etwa sehen, wie Sie die von Pascal her gewohnte Art von Vektoren (Arrays) mit beliebigem Anfangsindex in C selbst implementieren können. Dazu müssen Sie die Bedeutung des Operators [] für den Vektortyp neu definieren ("überladen"), damit Sie in gewohnter Weise Ausdrücke der Form A[i] für Ihren eigenen Datentyp verwenden können. Würde der Ausdruck A[i] einen Wert zurückliefern, dann wären aber Zuweisungen wie A[i] = b; ebenso sinnlos wie 3 = b;, denn einem Wert kann man natürlich nichts zuweisen. Dieses Problem läßt sich lösen, wenn A[i] statt des Wertes des i-ten Elements von A die entsprechende Referenz zurückliefert.

Zusammenfassung

- Der Operator & dient zur Deklaration von Referenzparametern und Referenzvariablen. Außerhalb von Variablen-Definitionen dient er als Adreßoperator.
- Bei der Definition einer Referenzvariablen wird durch die Initialisierung die Referenz hergestellt.

Überladen von Funktionsnamen

Nicht selten benötigt man verschiedene Varianten einer Funktion, die sich nur durch die übergebenen Typen unterscheiden. So könnte man etwa folgende Funktionen zur Berechnung des Absolutbetrags einer Zahl erklären:

```
int abs (int n) {
  if (n < 0) return -n;
  else return n;
```

```
}
double fabs (double x) {
  if (x < 0) return -x;
  else return x;
}
```

In C++ dürfen Sie beiden Funktionen den gleichen Namen geben (den Funktionsnamen *überladen*), um damit anzudeuten, daß es sich beide Male um den gleichen Vorgang handelt. Der Compiler sucht dann jeweils an Hand der Anzahl und der Typen der aktuellen Parameter die passende Version.

In der Standard-Funktionsbibliothek von C finden Sie übrigens die beiden Funktionen unter den Namen `abs` und `fabs` (und zusätzlich noch die Variante `labs` für Argumente vom Typ `long`). Das liegt daran, daß in C das Überladen von Funktionen noch nicht erlaubt war.

Sie werden vielleicht einwenden, daß es überladene Funktionen auch in Pascal gibt, denn die Pascal-Funktion `abs` ist auf Argumente beliebiger Zahltypen anwendbar. Das ist zwar richtig, aber gerade hier zeigt sich ein Schwachpunkt von Pascal: Die Sprache benutzt Mittel, deren Verwendung dem Anwendungsprogrammierer verwehrt ist: Gäbe es die Funktion `abs` nicht als Standardfunktion in Pascal, könnte man sie in Pascal nicht implementieren. Ebenso verhält es sich mit der Pascal-Prozedur `write`. Sie ist auf alle Standard-Datentypen anwendbar, nicht jedoch auf neu definierte Typen. So ergibt sich eine Zwei-Klassen-Sprache, in der den neu definierten Typen weniger Möglichkeiten offenstehen als den etablierten Typen.

Als weiteres Beispiel nehmen wir an, daß Sie öfter das Maximum von zwei oder drei Ganzzahlen bestimmen müssen. Sie könnten dann schreiben:

```
int max (int a, int b) {
  if (a > b) return a;
  else return b;
}

int max (int a, int b, int c) {
  int m;
  if (a > b) m = a;
  else m = b;
  if (c > m) m = c;
  return m;
}
```

Jetzt kann der Compiler die passende Funktion an Hand der Anzahl der aktuellen Parameter auswählen.

Beachten Sie, daß alle Varianten von überladenen Funktionen sich eindeutig an Hand der Anzahl oder Typen der Parameterliste unterscheiden müssen. Eine Unterscheidung allein im Typ des Funktionswerts ist nicht erlaubt.

Überladen von Funktionen ist immer dann sinnvoll, wenn mehrere Funktionen etwas bedeutungsgleiches mit verschiedenen Datentypen tun, etwa sie auf dem Bildschirm darstellen.

Vorgabe-Argumente

Beim Entwurf von Funktionen steht man oft vor der Wahl, ob die Funktion unkompliziert in der Anwendung oder universell verwendbar sein soll. Betrachten wir als Beispiel die Funktion, die Borland C++ zum Zeichnen von Ellipsen bietet:

```
void ellipse (int x, int y,
              int stangle, int endangle,
              int xradius, int yradius);
```

Die ersten beiden Parameter geben den Mittelpunkt der Ellipse an, die letzen beiden Parameter die beiden Halbachsen. Die Parameter `stangle` und `endangle` erlauben, einen Teil der Ellipse zu zeichnen: Sie geben den Anfangs- und Endwinkel (in Altgrad) an. Für den Normalfall einer vollständige Ellipse müssen hier stets die Werte 0 und 360 eingesetzt werden. Die Funktion wäre leichter zu merken, wenn man diese beiden Werte weglassen könnte. Deshalb ist es in C++ erlaubt, bei der Deklaration einer Funktion Argumenten Vorgabewerte zu geben, die automatisch verwendet werden, wenn die Argumente weggelassen werden. Damit der Compiler eindeutig erkennen kann, welche Parameter weggelassen wurden, gilt die Einschränkung, daß solche Vorgabe-Argumente am Ende der Parameterliste stehen müssen. Die Vorgabeargumente dürfen nur bei der ersten Deklaration angegeben werden!

Als Beispiel schreiben wir eine Funktion (mit dem doppeldeutigen Namen `Ellipse`), bei der die Winkelargumente weggelassen werden können:

```
void Ellipse (int x, int y, int a, int b, int w0=0, int w1=360) {
  ellipse (x, y, w0, w1, a, b);
}
```

Dann werden durch die beiden Aufrufe

```
  Ellipse (40, 40, 30, 20);
  Ellipse (80, 40, 30, 20, 0, 90);
```

eine volle und eine viertel Ellipse gezeichnet.

Übungsaufgabe 3:

Schreiben Sie eine Funktion `max`, die das Maximum von zwei, drei oder vier `int`-Werten ermittelt!

Inline-Funktionen

Wenn Sie in Pascal sehr einfache und kurze Programmstücke als Prozedur oder Funktion implementieren, erkaufen Sie sich die bessere Lesbarkeit des Programms durch Verminderung der Programmgeschwindigkeit. In einem zeitkritischen Programm könnten Sie in Versuchung geraten, anstelle von einfachen Funktionsaufrufen wie

```
// Version A: Funktionsaufruf:
int max (int i, int j) {
  if (i > j) return i; else return j;
}
...
  m = max (a, b);
```

direkt den Inhalt der Funktionsdefinition zu schreiben:

```
// Version B: Direkte Implementierung:
if (a > b) m = a; else m = b;
```

Um Ihnen einen Anhaltspunkt für solche Entscheidungen zu geben, habe ich die Codestücke, die der Compiler von Borland C++ (für den 80286-Prozessor) erzeugt, miteinander verglichen.

Der Code für den Funktionsaufruf (Version A) belegt 10 Bytes. Dazu kommen einmalig noch 22 Bytes für den Code der Funktion `max`. Die Version B benötigt 13 Bytes. Hier gibt es keinen eindeutigen Gewinner: Bei bis zu 7 Aufrufen von `max` ist die Version B platzsparender, ab dem 8. Aufruf liegt die Version A vorn.

Ganz anders sieht es allerdings bei der Geschwindigkeit aus: Version A benötigt 32 Prozessortakte für den Funktionsaufruf und 57 oder 59 Takte (je nach Ergebnis des Vergleichs) für die Abarbeitung der Funktion, insgesamt also rund 90 Takte. Die Version B begnügt sich mit nur 15 oder 18 Takten, ist also 5 bis 6 mal schneller.

Dieser gewaltige Unterschied erklärt sich dadurch, daß vor einem Funktionsaufruf die Parameter auf den Stack gebracht werden müssen, die Kontrolle an die Funktion übergeben und anschließend die Parameter wieder vom Stack genommen werden müssen. So kommt es, daß bei einer so einfachen Funktion wie der `max`-Funktion dieser rein organisatorische Aufwand über viermal soviel Zeit benötigt, wie die eigentliche Rechnung.

C++ bietet Ihnen glücklicherweise die Möglichkeit, die Lesbarkeit der Version A mit der Effizienz der Version B zu vereinen: Sie können eine Funktion als *inline-Funktion* deklarieren, indem Sie ihr das Schlüsselwort `inline` voranstellen. Damit wird der Compiler angewiesen, jeden Funktionsaufruf so zu codieren, als ob er direkt implementiert wäre. Damit können wir also eine dritte Version schreiben:

```
// Version C: inline-Funktion:
inline int max (int i, int j) {
  if (i > j) return i; else return j;}
...
m = max (a, b);
```

Diese Version erzeugt den gleichen Code wie die Version B. Für das Beispiel der `max`-Funktion ist das die optimale Lösung. Wegen der fehlenden Parameterübergabe sind `inline`-Funktionen immer schneller als normale Funktionen. Sie sollten allerdings nur extrem kurze Funktionen als `inline` deklarieren, denn bei längeren `inline`-Funktionen, die oft im Programmtext verwendet werden, macht sich der erhöhte Speicherbedarf schnell bemerkbar. Das Schlüsselwort `inline` ist übrigens keine Garantie, daß die

Funktion auch wirklich als `inline`-Funktion übersetzt wird. Borland C++ ignoriert es bei zu komplizierten Funktionen. Zur Erleichterung der Arbeit mit dem Debugger gibt es eine Option, die das Schlüsselwort `inline` wirkungslos macht.

1.9 Typkonvertierungen

Implizite Konvertierungen

C++ ist bei der Verwendung der elementaren Typen in Ausdrücken, Zuweisungen und Prozeduraufrufen recht liberal. Im Gegensatz etwa zu Modula-2, wo jede Konvertierung ausdrücklich vom Programmierer angefordert werden muß, nimmt C++ eine ganze Menge von Standard-Konvertierungen vor.

Innerhalb von Ausdrücken dürfen die vordefinierten Datentypen im Prinzip beliebig gemischt werden. Hiervon sollten Sie allerdings möglichst mit Maßen Gebrauch machen, denn bei der Konvertierung können, zum Beispiel durch Rundungsfehler oder Abschneiden, Daten verfälscht werden. Deshalb sollten Sie sich über die Wirkung automatischer Konvertierungen im Klaren sein.

Stehen links und rechts eines binären Operators Operanden verschiedener Typen, geht C++ so vor:

- **1.** Operanden vom Typ `char` und `short` werden in `int` umgewandelt.
- **2.** Operanden vom Typ `float` werden in `double` umgewandelt.
- Jetzt enthält der Ausdruck nur noch Operanden der Typen
  - `long double`
  - `double`
  - `unsigned long`
  - `long`
  - `unsigned`
  - `int`
- **3.** Der niedrigere der beiden Operanden wird in den höheren Typ umgewandelt. Das Ergebnis ist dann vom höheren Typ.

Bei Zuweisungen wird automatisch der Wert der rechten Seite in den Typ der linken Seite umgewandelt. Ebenso wird bei der Übergabe von Wertparametern an Prozeduren verfahren. In beiden Fällen können Werte verfälscht werden, wenn der Zieltyp das korrekte Ergebnis nicht aufnehmen kann. Viele Compiler weisen durch Warnungen auf solche Gefahren hin.

Bei der Konvertierung von einem ganzzahligen in einen anderen ganzzahligen Typ wird so verfahren:

- **gleichbleibende Länge**:
  Der Inhalt wird unverändert übernommen. Er kann allerdings anders interpretiert werden. Beispiel:

```
unsigned u = 0xFFFF; // u = 65535
int i = u; // i = -1 (Zweierkomplement!)
```

- **Ziel kleiner als Quelle**:
  Die Quelle wird durch Abschneiden der höherwertigen Bits auf die gewünschte Länge gebracht.
- **Ziel größer als Quelle**:
  - Quelle signed: Die höherwertigen Stellen werden mit dem Vorzeichenbit aufgefüllt.
  - Quelle unsigned: Die höherwertigen Stellen werden mit Nullen aufgefüllt.

Explizite Konvertierungen

Sie können auch explizit eine Typumwandlung erzwingen, indem Sie den Namen des Zieltyps wie einen Funktionsaufruf vor den ursprünglichen Ausdruck setzen:

```
long l;
int i;
...
i = int (l);
i = l;
```

Die Umwandlung von l in den Typ int wird hier in der vorletzten Zeile explizit angefordert. In der letzten Zeile wird die Konvertierung automatisch ausgeführt, allerdings mit einer Warnung des Compilers, da die Umwandlung von l nach int Unsinn ergibt, wenn der Wert von l zu groß ist.

Bei der expliziten Konvertierung erhalten Sie keine Warnung. Sie geben dem Compiler die Blanko-Vollmacht, Ihre Zahl notfalls zu verstümmeln.

Manchmal kann eine explizite Konvertierung den Compiler auch vor überflüssiger Arbeit bewahren:

```
int i, n=1;
double x = 1.2;
i = n + x;
i = n + int (x);
```

In der vorletzten Zeile wird zunächst n nach double konvertiert, anschließend die Summe mit Gleitkomma-Arithmetik errechnet und zum int-Wert 2 konvertiert.

Die letzte Zeile kommt zum gleichen Ergebnis mit nur einer Konvertierung und (schnellerer) Ganzzahl-Arithmetik.

Der cast-Operator

Die soeben beschriebene explizite Konvertierung ist nur mit Typnamen möglich. Wollen Sie etwa einen Zeiger p in einen Zeiger auf char umwandeln, so müssen Sie zunächst diesem

Typ einen Namen geben:

```
typedef char *charptr;
```

Jetzt läßt sich die Umwandlung so schreiben: `charptr (p)`.

Wenn Sie nicht jedem Typ einen Namen geben wollen, können Sie auch eine alternative Form der expliziten Typumwandlung verwenden, den *cast-Operator* (In Standard-C gibt es übrigens nur den `cast`-Operator): Der Zieltyp wird in runde Klammern gesetzt und vor den Ausdruck geschrieben. Im letzten Beispiel könnte man also schreiben: `(char *) p`.

Vereindeutigung von Funktionsaufrufen

Die Möglichkeiten des Überladens von Funktionen sowie der Verwendung von Vorgabe-Argumenten können in Extremfällen zu schwer durchschaubaren Konflikten führen.

Betrachten Sie folgendes Beispiel:

```
#include <iostream.h>

char f (int a) {return 'A';}
char g (int a, int b=0) {return 'B';}
char g (long a) {return 'C';}

main () {
  cout << f(1L) << g(1,2) << g(1) << g(1L) << g(1L,2);
}
```

Was geschieht bei den fünf Funktionsaufrufen im Hauptprogramm?

- `f(1L)`:
  Die Funktion `f` verlangt ein Argument vom Typ `int`. Das übergebene Argument vom Typ `long` wird automatisch nach `int` konvertiert. Der Funktionswert ist also `'A'`.

- `g(1,2)`:
  Die Funktion `g` ist überladen. Zunächst prüft der Compiler, ob die Parametertypen einer Version exakt passen. Dies ist bei der ersten Version der Fall. Der Funktionswert ist `'B'`.

- `g(1)`:
  Auch hier wird eine exakte Typübereinstimmung mit der ersten Version gefunden. Für den nicht übergebenen zweiten Parameter wird der Vorgabewert eingesetzt. Der Funktionswert ist `'B'`.

- `g(1L)`:
  Der Parameter vom Typ `long` paßt auf die zweite Version. Der Funktionswert ist `'C'`.

- `g(1L,2)`:

Von der Anzahl der Parameter her kommt nur die erste Version von g in Frage. Der erste Parameter wird nach int konvertiert. Der Funktionswert ist 'B'.

Beachten Sie, daß die Aufrufe g(1) und g(1L) bei beiden Versionen von g korrekt wären, wenn es die jeweils andere nicht gäbe. Sind beide vorhanden, sucht sich der Compiler die passendere heraus nach folgendem (etwas vereinfachten) Schema:

Vereindeutigung eines Aufrufs überladener Funktionen:

- **1.** Stimmt die Parameterliste einer Version in Anzahl und Typ mit den Argumenten überein, so wird sie ausgewählt. Da sich alle Varianten von überladenen Funktionen eindeutig an Hand der Anzahl oder Typen der Parameterliste unterscheiden müssen, kann es höchstens eine solche Variante geben.
- **2.** Gibt es beim ersten Schritt keine exakte Übereinstimmung, werden folgende Konvertierungen der Argumente vorgenommmen:

  char nach int
  unsigned char nach int
  float nach double.

  Jetzt wird wieder wie im ersten Schritt auf exakte Übereinstimmung geprüft.
- **3.** Gibt es beim zweiten Schritt immer noch keine exakte Übereinstimmung, wird versucht, durch Standard-Konvertierungen eine Übereinstimmung herzustellen. Gibt es hier mehrere Möglichkeiten, wird dies vom Compiler als Fehler eingestuft.

Viele Probleme lassen sich vermeiden, wenn man die gewünschten Konvertierungen explizit vorgibt. Dehalb ist es nicht unbedingt nötig, daß Sie das Schema zur Vereindeutigung beherrschen. Am folgenden Beispiel können Sie sich die Wirkung der Regeln klarmachen.

```
#include <iostream.h>

void f (int a) {cout << 'A';}
void f (float a) {cout << 'B';}

main () {
  int i; unsigned u; float x; double y; char c;
              // Schritt  Ausgabe
  f (i);      // 1.       'A'
  f (x);      // 1.       'B'
  f (c);      // 2.       'A'
  f (u);      // 3.       Fehler: Mehrdeutigkeit (float/int)
  f (int(u)); // 1.       'A'
  f (y);      // 3.       Fehler: Mehrdeutigkeit (float/int)
}
```

Übungsaufgabe 4:

Entfernen Sie die fehlerhaften Zeilen aus dem folgenden Programm! Was gibt das Programm jetzt aus?

```
#include <iostream.h>
```

```
void f (int a, long b) {cout << 'A';}
void f (long a, int b) {cout << 'B';}

main () {
  int i;
  long l;
  f (i,l);
  f (l,i);'
  f (i,i);
  f (l,l);
  f (i, long(i));
  f (l, int(i));
}
```

1.10 Abgeleitete Datentypen

Strukturen

C++ erlaubt ebenso wie Pascal die Definition von Verbundtypen, nämlich Datentypen, die aus mehreren Elementen (Komponenten) fester, aber beliebiger Typen zusammengesetzt sind. Ein einfaches Beispiel zeigt die syntaktischen Unterschiede:

```
type Stadt =                            struct Stadt {
  record
    Name     : string[40];                char Name[41];
    PLZ      : integer;                   int  PLZ;
    Einwohner: longint;                   long Einwohner;
  end;                                  };

var s1, s2: Stadt;                      Stadt s1, s2;
```

Beachten Sie bitte folgenden wesentlichen Unterschied: In Pascal wird mit der `record`-Konstruktion ein namenloser Datentyp gebildet. Dieser kann entweder direkt einer Variablen zugeordnet oder zur Deklaration eines Typnamens verwendet werden. In dieser Weise mehrfach definierte Typen sind nicht kompatibel. In C++ wird mit der `struct`-Konstruktion ein neuer Datentyp deklariert, der im Rest der Übersetzungseinheit Gültigkeit hat. Daher entfallen die Inkompatibilitätsprobleme. Gleichzeitig mit der Struktur-Deklaration können auch Variablen dieses Typs definiert werden. Werden mit der Typdefinition keine Variablen definiert, muß trotzdem nach der schließenden geschweiften Klammer ein Semikolon stehen! Es folgt ein vergleichendes Beispiel:

```
type komplex = record x, y: real end;
var a  : komplex;                       struct komplex {float x, y;} a;
    b,c: record x, y: real end;
    d  : record x, y: real end;         komplex b;
begin
  a := b;  (* Type mismatch *)          main () {
  b := c;                                 a = b;
  b := d;  (* Type mismatch *)          }
end.
```

Strukturen sind in C++ Spezialfälle der Klassen, auf denen die objektorientierte Programmierung aufbaut.

Varianten (Unions)

Ein Sprachmittel von Pascal, das in der Praxis meist zweckentfremdet eingesetzt wird, ist der *variante Record*. Er ist dafür vorgesehen, daß am Ende eines Records, abhängig vom Inhalt einer Selektor-Komponente verschiedenartige Daten am gleichen Platz gespeichert werden können, um somit Speicherplatz zu sparen. Diese Konstruktion ist insofern etwas problematisch, als damit die Typstrenge von Pascal unterlaufen wird: Der Compiler kann das korrekte Ansprechen der varianten Komponenten nämlich nicht überprüfen.

In maschinennahen Programmen möchte man manchmal auch bewußt bestimmte Daten unter verschiedenen Aspekten interpretieren. Da es in Pascal hierzu kein offizielles Sprachmittel gibt, wird zu diesem Zweck oft der variante Record mißbraucht. Tatsächlich ist es sogar erlaubt, die in diesem Fall überflüssige Selektorvariable ganz wegzulassen, wie das folgende Beispiel zeigt:

```
var doppelt: record case integer of
               1: (u: Word);
               2: (i: integer);
             end;
begin
  doppelt.i := -1;
  writeln (doppelt.u:8);           (* Ergebnis: 65535 *)
end.
```

In Turbo Pascal gibt es für solche Zwecke eine direktere Lösung:

```
var u: Word;
    i: integer absolute u;
begin
  i := -1;
  write (u:8);
end.
```

C++ bietet zum gleichen Zweck die `union`-Konstruktion an. Eine `union` hat die gleiche syntaktische Struktur wie eine `struct`. Während aber alle `struct`-Komponenten im Speicher hintereinander angelegt werden, beginnen alle `union`-Komponenten an der gleichen Stelle. Der Speicherplatz einer `union` ist also gleich dem Speicherplatz seiner größten Komponente. Damit läßt sich das Beispiel in C++ so schreiben:

```
#include <iostream.h>
main () {
  union doppelt {int i; unsigned u;} d;
  d.i = -1;
  cout << d.u;                       // Ergebnis: 65535
}
```

Natürlich läßt sich auch ein orthodoxer varianter Record in C++ direkt nachbilden: Ihm entspricht eine `struct`, deren letzte Komponente eine `union` ist.

Für unseren einfachen Spezialfall bietet sich in C++ eine sogenannte *anonyme union* an. Dabei wird kein Name für die `union` vergeben. Die Komponenten dürfen dann nicht mit anderen Namen ihres Geltungsbereichs kollidieren und können direkt angesprochen

werden:

```
#include <iostream.h>
main () {
  union {int i; unsigned u;};
  i = -1;
  cout << u;                            // Ergebnis: 65535
}
```

Aufzählungstypen

Aufzählungstypen sind ein sehr nützliches Mittel, implementierungsinterne Codierungen zu verbergen und damit Programme lesbarer zu machen. In Pascal könnte man etwa die 16 Standard-Farben des EGA-Adapters so definieren:

```
type Farbe = (
  BLACK,    BLUE,         GREEN,      CYAN,
  RED,      MAGENTA,      BROWN,      LIGHTGRAY,
  DARKGRAY, LIGHTBLUE,    LIGHTGREEN, LIGHTCYAN,
  LIGHTRED, LIGHTMAGENTA, YELLOW,     WHITE);
```

Die so definierten Namen haben die Werte 0 bis 15. Wenn wir allerdings jetzt zum Beispiel eine Farbe setzen wollen mit

```
setcolor (BLUE);   (* Type mismatch *)
```

bekommen wir eine Fehlermeldung ("Type mismatch"). In der Unit Graph sind die Farben nämlich nicht als Aufzählungstyp, sondern als Konstanten vom Typ `word` definiert. Die Prozedur `setcolor` verlangt ein Argument vom Typ `word` und nicht vom Typ `Farbe`. Obgleich die Elemente unseres Aufzählungstyps die gleichen Werte repräsentieren, sind sie nicht typverträglich mit den `word`-Konstanten. Wir können die Typanpassung allerdings explizit verlangen:

```
setcolor (word (BLUE));
```

Der soeben definierte Aufzählungstyp sieht in C++ ganz ähnlich aus:

```
enum Farbe = {
  BLACK,    BLUE,         GREEN,      CYAN,
  RED,      MAGENTA,      BROWN,      LIGHTGRAY,
  DARKGRAY, LIGHTBLUE,    LIGHTGREEN, LIGHTCYAN,
  LIGHTRED, LIGHTMAGENTA, YELLOW,     WHITE};
```

Der wesentliche Unterschied besteht in der etwas geringeren Typstrenge von C++: Ein mit `enum` definierter Typ wird zwar vom Typ `int` unterschieden, aber es gibt implizite Konvertierungen in beiden Richtungen zwischen jedem `enum`-Typ und `int`. Da ein `enum`-Typ nur `int`-Werte enthalten kann, ist eine Konvertierung von `enum` nach `int` unproblematisch. In der umgekehrten Richtung ergibt nicht jede `int`-Zahl, in einen `enum`-Typ umgewandelt, einen Sinn. Deshalb verlangt das Handbuch von Borland C++, daß einer `enum`-Variablen nur Werte der gleichen Aufzählung zugewiesen werden dürfen. Tatsächlich begnügt sich der

Compiler mit einer Warnung (Der Zortech-Compiler gibt hier eine Fehlermeldung aus).

Im Normalfall werden den Elementen eines Aufzählungstyps aufeinanderfolgende Werte ab Null zugewiesen. Manchmal ist es aus technischen Gründen nötig, den Elementen bestimmte, nicht unbedingt aufeinanderfolgende Werte zu geben. Deshalb ist es erlaubt, an einzelne Elemente explizite Werte zu vergeben:

```
enum kleinePrimzahl {ZWEI=2, DREI, FUENF=5};
```

Elemente, denen kein Wert zugewiesen wurde, bekommen den um eins erhöhten Wert ihres Vorgängers (das Erste Element den Wert Null). In der Praxis sollten Sie nur Werte in einer Aufzählung zusammenfassen, die auch irgendwie logisch zusammengehören.

Neben den enum-Typen lassen sich auch *unbenannte Aufzählungen* definieren, indem man einfach den Namen wegläßt:

```
enum {SCHWARZ, WEISS};
```

Unbenannte Aufzählungen definieren keinen eigenen Typ, sondern stellen eine Sammlung von int-Konstanten bereit. Sie können so zum Beispiel die von Pascal gewohnten Wahrheitswertkonstanten definieren:

```
enum {false, true};
```

Naheliegend wäre auch die folgende Definition des Datentyps boolean:

```
enum boolean {false, true};
```

Da aber die logischen Operatoren in C++ int-Werte liefern, bekommt man so Verträglichkeitsprobleme. Man sollte also die unbenannte Aufzählung zusammen mit der Definition

```
typedef int boolean;
```

verwenden. Das folgende Beispiel zeigt nochmals den Umgang mit Aufzählungstypen:

```
enum FARBE {ROT, GRUEN};  // benannte enum: eigener Typ
enum SPRACHE {C, PASCAL}; // dto.
enum {false, true};       // unbenannte enum ist int
typedef int boolean;

int main () {
  FARBE   f;
  SPRACHE s;
  boolean b;

  f = s;     // nicht erlaubt (in Borland C++ nur Warnung)
  f = b;     // nicht erlaubt (in Borland C++ nur Warnung)
  f = true;  // nicht erlaubt (in Borland C++ nur Warnung)
  b = f;     // erlaubt!
  b = false; // erlaubt
}
```

Bitfelder

In der systemnahen Programmierung gibt es oft eine Fülle von Variablen zu verwalten, deren Wertebereich sehr klein ist, typischerweise logische Werte (*Flags*). Um hier nicht für jede Variable ein Byte oder gar Wort zu verschwenden, werden mehrere Informationen in ein Byte gepackt. Typische Beispiele sind die Variablen im Systembereich Ihres PCs. So gibt es etwa an der Speicheradresse 417H ein Byte, das in seinen einzelnen Bits ständig den Zustand der Umschalttasten sowie der Strg- und Alt-Tasten widerspiegelt.

Da die kleinste adressierbare Einheit Ihres Rechners das Byte ist, sind schon einige Verrenkungen nötig, um einzelne Bits zu manipulieren. Zum Prüfen eines Bits kann man etwa eine bitweise Und-Verknüfung mit einer Konstanten vornehmen, die genau an der richtigen Stelle ein gesetztes Bit hat, und prüfen, ob das Ergebnis von Null verschieden ist. Solche Kunststücke machen das Lesen eines Programms allerdings zur Qual und sollten nach Möglichkeit vermieden werden.

In Pascal bietet sich zur Lösung solcher Probleme das Mengenkonzept an. So können etwa 16 durch einen Aufzählungstyp definierte Flags in einem Maschinenwort untergebracht werden.

Wir werden später sehen, wie sich solche Probleme mit Hilfe des Klassenkonzepts in C++ elegant lösen lassen. Deshalb können Sie diesen Abschnitt ohne Schaden überspringen.

Aus Standard-C steht die Möglichkeit zur Verfügung, innerhalb von Strukturen *Bitfelder* zu definieren. Das sind Ganzzahl-Komponenten, die eine Breite von 1 bis 16 Bits haben. Im folgenden Beispiel teilen sich drei Bitfeld-Komponenten ein Maschinenwort:

```
struct Beispiel1 {
  int      i:6; // Wertebereich: -32 ... 31
  unsigned u:3; // Wertebereich:   0 ...  7
  unsigned j:7; // Wertebereich:   0 ...127
};
```

Zum Auffüllen von Lücken können auch *unbenannte Bitfelder* verwendet werden:

```
struct Beispiel2 {
  unsigned i:4;
  unsigned  :3;  // nicht ansprechbare Lücke von 3 Bits
  unsigned j:9;
};
```

Falls Sie Bitfelder verwenden wollen, sollten Sie folgende Hinweise beherzigen:

- Es gibt keine Vorschrift, wie ein Compiler die einzelnen Bitfelder einer Struktur im Speicher anordnen soll. Wenn Sie Daten verarbeiten, die nicht von Ihrem Programm erzeugt worden sind, können Sie sich nicht darauf verlassen, daß ein anderer Compiler die Daten ebenso interpretiert. In diese Abhängigkeit vom verwendeten Compiler begeben Sie sich zum Beispiel, wenn Sie einem Zeiger auf eine geeignete Struktur mit Bitfeldern die Adresse der Speicherstelle mit den Tastenflags

zuweisen.

- Die Sprachdefinition schreibt lediglich vor, daß unsigned-Bitfelder unterstützt werden müssen. Bitfelder vom Typ int (in Zweierkomplement-Darstellung) können, müssen aber nicht unterstützt werden. Borland C++ unterstützt int-Bitfelder, Zortech C++ nicht!
- Bitfelder können nicht Elemente einer Union sein.
- Da Bitfelder im allgemeinen nicht auf Byte-Grenzen beginnen, darf der Adreß-Operator nicht auf Bitfelder angewendet werden.

Zeigertypen und Vektoren

Auch wenn Sie in Pascal bisher die Verwendung von Zeigern gemieden haben: in C++ können Sie nicht erfolgreich programmieren, ohne sich um das Zeigerkonzept zu kümmern. Das liegt daran, daß es in C++ keine Arrays im Sinne von Pascal gibt (Wir werden die Pascal-Arrays später mit Hilfe des Klassenkonzepts nachbilden).

Ein *Zeiger* (*Pointer*) ist zunächst einmal eine Speicheradresse. Dazu trägt er aber noch eine Typinformation. Diese wird zum Zugriff auf das Objekt an der Zeigeradresse (*Dereferenzierung*) benötigt. So können etwa ein char-Zeiger und ein int-Zeiger auf die gleiche Adresse, aber auf verschiedene Objekte zeigen.

Zeiger werden benötigt, wenn während der Laufzeit eines Programms dynamisch Speicherplatz vergeben wird, so etwa zum Aufbau von Bäumen oder verketteten Listen. Die folgende Gegenüberstellung zeigt die Unterschiede in der Bezeichnung und Dereferenzierung von Zeigern:

Zeiger in Pascal

```
var p: ^integer;
...
new (p);
p^ := 1;
dispose (p);
```

Zeiger in C++

```
int *p;
...
p = new int;
*p = 1;
delete p;
```

Es folgt eine systematische Gegenüberstellung:

Zeigertypen	**Pascal**	**C++**
Deklaration	^**Typ**	***Variable**
Dereferenzierung	**Variable**^	***Variable**
Speicherreservierung	new (**Variable**)	**Variable** = new **Typ**;
Speicherfreigabe	dispose (**Variable**)	delete **Variable**;

Bei der Deklaration einer Zeigervariable in C++ kann man natürlich statt int *p; auch int* p; schreiben, um anzudeuten, daß die Variable p heißt und der Typ Zeiger auf int ist. Wenn Sie aber mehrere Zeigervariablen hintereinander erklären, müssen Sie

aufpassen: `int* p1, p2;` deklariert nicht etwa zwei Zeigervariablen, wie die Schreibweise suggeriert, sondern eine Zeigervariable p1 und eine int-Variable p2. Hier müßten Sie `int *p1, *p2;` schreiben!

new Typ reserviert Speicherplatz für ein Objekt des Datentyps Typ und liefert einen Zeiger auf dieses Objekt. Dies kann dann einem entsprechenden Zeiger zugewiesen werden. Beispiel:

```
int *ptr;
ptr = new int;
```

Wahlweise kann man mit Hilfe des new-Operators auch ein Vielfaches des Speicherplatzes für den Zeigertyp anfordern. In der Anweisung

```
ptr = new int [4];
```

reserviert new zusammenhängenden Speicher für vier int-Zahlen und liefert einen Zeiger auf den Anfang dieses Speicherplatzes zurück. Nach der Zuweisung zeigt ptr also auf die erste der vier int-Zahlen. Wie können aber die übrigen Zahlen angesprochen werden? Erinnern Sie sich: In Pascal sind alle arithmetischen Operationen mit Zeigern verboten. Welchen Sinn hätte auch etwa das Produkt zweier Zeiger?

Um die soeben definierten vier int-Objekte ansprechen zu können, wäre es hilfreich, wenn man Zeigeradressen erhöhen könnte, etwa so:

```
ptr1 = ptr + sizeof (int);       // hypothetische Zeigerarithmetik
ptr2 = ptr + 2 * sizeof (int);
ptr3 = ptr + 3 * sizeof (int);
```

C++ erlaubt tatsächlich die Addition oder Subtraktion von int-Zahlen zu Zeigertypen. Um das umständliche und fehleranfällige Multiplizieren mit der Typgröße zu vermeiden, rechnet C++ hierbei automatisch in Vielfachen der Typgröße und nicht in Bytes! Die Zuweisung

```
ptr3 = ptr + 3;
```

addiert zur Zeigeradresse von ptr dreimal die Typgröße (von int), also zum Beispiel 6 Bytes. Wir können also die drei zusätzlichen Objekte so ansprechen:

```
ptr1 = ptr + 1;       // Zeigerarithmetik in C++
ptr2 = ptr + 2;
ptr3 = ptr + 3;
```

Additionen und Subtraktionen zu Zeigern sind auch mit Hilfe der Operatoren ++, --, += und -= erlaubt. Außerdem kann man die Differenz zweier Zeiger gleichen Typs berechnen. Das Ergebnis ist eine int-Zahl. Sie gibt die Differenz der Zeiger-Adressen in Typ-Einheiten an.

Wir wollen jetzt sehen, wie wir die Array-Behandlung von Pascal nach C++ übertragen können:

Array in Pascal

```
program ArrayTest;
var
  i: integer;
  A: array[0..3] of integer;
begin

  for i := 0 to 3 do
    A[i] := i;

end.
```

Array in C++ (1. Version)

```
main ()
{
  int i;
  int *A;

  A = new int [4];
  for (i=0; i<4; i++)
    *(A+i) = i;
  delete A;
}
```

Glücklicherweise gibt es alternativ eine besser lesbare Schreibweise. Ist `A` ein Zeiger und `i` eine `int`-Zahl, so darf man statt `*(A+i)` auch `A[i]` schreiben:

Array in Pascal

```
program ArrayTest;
var
  i: integer;
  A: array[0..3] of integer;
begin

  for i := 0 to 3 do
    A[i] := i;

end.
```

Array in C++ (2. Version)

```
main ()
{
  int i;
  int *A;

  A = new int [4];
  for (i=0; i<4; i++)
    A[i] = i;
  delete A;
}
```

Das folgende Diagramm verdeutlicht die Möglichkeiten der Dereferenzierung von Zeiger-Objekten:

Jetzt stört noch, daß wir den Speicherplatz explizit belegen und zum Schluß wieder freigeben müssen. In Pascal geschieht dies automatisch: der Speicherplatz wird mit der Deklaration reserviert und bei Verlassen des Gültigkeitsbereichs automatisch freigegeben. Dafür muß die Größe ein konstanter Ausdruck sein, während es in C++ auch möglich ist, die Größe erst zur Laufzeit zu bestimmen. Von dieser neuen Freiheit haben wir in unserem Beispiel gar keinen Gebrauch gemacht. In solchen Fällen können wir uns noch weiter der Pascal-Schreibweise nähern:

Array in Pascal

```
program ArrayTest;
var
  i: integer;
  A: array[0..3] of integer;
begin
  for i := 0 to 3 do
    A[i] := i;
end.
```

Array in C++ (3. Version)

```
main ()
{
  int i;
  int A[4];

  for (i=0; i<4; i++)
    A[i] = i;
}
```

Durch eine Deklaration der Form

Vektor-Deklaration:
Typname Variablenname [**int-Konstante**];

wird Speicherplatz für **int-Konstante** Objekte von **Typname** reserviert und **Variablenname** mit der Adresse dieses Speicherplatzes initialisiert. Der Zeiger **Variablenname** wird als Konstante aufgefaßt. Beim Verlassen des Gültigkeitsbereichs wird der so belegte Speicher automatisch freigegeben.

Obgleich die letzte Version schon fast wie in Pascal aussieht, sollten Sie sich immer wieder vergegenwärtigen, daß es eigentlich keine Arrays in C++ gibt. Was geschieht etwa in folgendem Beispiel?

```
main ()
{
  int A[4], B[4];
  ...
  A = B;
}
```

Da A und B Zeiger sind, hätte die Zuweisung die Wirkung, daß A auf die gleiche Adresse wie B zeigt. Der für A ursprünglich reservierte Speicherplatz wäre nun unerreichbar. Tatsächlich verbietet der Compiler hier die Zuweisung, weil A als konstantem Zeiger kein Wert zugewiesen werden kann. Das vielleicht beabsichtigte Kopieren könnte man stattdessen etwa so formulieren:

```
for (i=0; i<4; i++) A[i] = B[i];
```

Wie Sie gesehen haben, gibt es zu jedem Datentyp in C++ einen entsprechenden Zeigertyp. Daneben gibt es den *universellen Zeigertyp* `void *`. Einer Variablen dieses Typs dürfen Sie jeden beliebigen Zeiger zuweisen. In der umgekehrten Richtung müssen Sie die Typumwandlung explizit angeben:

```
void *univZeiger;
int  *intZeiger;

univZeiger = intZeiger;
intZeiger  = (int *) univZeiger; // korrekte Typumwandlung
intZeiger  = univZeiger;         // Syntaxfehler!
```

Universelle Zeiger können nicht dereferenziert werden, weil ihnen ja kein Datentyp zugeordnet ist. Ein Beispiel für die Anwendung von universellen Zeigertypen ist die Bibliotheksfunktion `memcpy`:

```
void *memcpy (void* dest, const void *src, size_t n);
```

`memcpy` kopiert `n` Bytes ab der Adresse `src` an die Adresse `dest`. Dabei dürfen sich Ziel und Quelle nicht überlappen. Der Datentyp `size_t` ist ein maschinenabhängig definierter Ganzzahltyp. Als Funktionswert wird der Zeiger `dest` zurückgegeben. Wir könnten die Funktion etwa so nachimplementieren:

```
void *memcpy (void *dest, const void *src, size_t n)
{
  size_t i;
  char *Ziel;
  char *Quelle;

  Ziel   = (char *) dest;
  Quelle = (char *) src;
  for (i=0; i<n; i++) Ziel[i] = Quelle[i];
  return dest;
}
```

Um jede beliebige Anzahl von Bytes kopieren zu können, haben wir die universellen Zeiger hier in `char`-Zeiger umgewandelt (Der Typ `char` belegt in jeder Implementierung genau ein Byte). Damit können wir das oben formulierte Kopieren von Vektoren kurz so schreiben:

```
memcpy (A, B, 4 * sizeof (int));
```

Vielleicht fragen Sie sich, wie man in C++ Vektoren darstellen kann, deren Indizes nicht bei Null anfangen. Eine etwas undurchsichtige Lösung dieses Problems ist die Verschiebung eines Zeigers auf den Vektoranfang:

```
var A: array[n0..n1] of integer;       int A0[n1-n0+1];
                                       int *A = A0 - n0;
```

Für eine saubere Lösung mit Indexüberprüfung müssen Sie sich bis zur Einführung des Klassenkonzepts gedulden.

In Pascal gibt es eine spezielle Zeigerkonstante `nil`, die auf kein Objekt zeigt und mit jedem Zeigertyp verträglich ist. Diese Konstante `nil` hat intern den Wert Null. In C++ ist es ganz ähnlich: Obwohl Zeigertypen nicht mit dem Typ `int` verträglich sind, kann die `int`-Konstante Null jedem Zeigertyp zugewiesen werden. Man sollte hierfür aber besser die in der Datei `stddef.h` vordefinierte Konstante `NULL` verwenden. Das ist lesbarer, und erhöht die Chance, mit zukünftigen Sprachänderungen verträglich zu bleiben.

Übungsaufgabe 5:

Warum ergibt die doppelte Zuweisung im folgenden Programm einen Syntaxfehler, während die beiden einfachen Zuweisungen korrekt sind?

```
#include <stddef.h>
int  *p1;
long *p2;

main () {
  p1 = NULL;
  p2 = NULL;
  p1 = p2 = NULL;
}
```

Zeichenketten (Strings)

Die Behandlung von Strings unterscheidet sich in C++ grundsätzlich von der in Turbo

Pascal. In Turbo Pascal beginnen Strings mit einem Byte, das die aktuelle Längenangabe enthält. Deshalb können Strings nicht länger als 255 Bytes werden. In C++ sind Strings im Prinzip char-Vektoren und dürfen beliebig lang werden. Wenn Sie eine String-Konstante definieren, hängt der Compiler automatisch ein zusätzliches Byte mit dem Wert Null an. Alle Standardfunktionen zur String-Behandlung halten sich ebenfalls an die Konvention, daß das Ende eines Strings durch ein Nullbyte angezeigt wird.

Wenn Sie beim Programmieren mit den String-Funktionen von Turbo Pascal Fehler machen, wie etwa im folgenden Programm, sind die Auswirkungen gewöhnlich ziemlich harmlos.

```
program StringFehler;
var s1, s2: string[10];
begin
  s1 := 'String-';
  s2 := 'Verkettung';
  s1 := concat (s1, s2);
  writeln (s1);
end.
```

Die Verkettung von s1 und s2 ergibt String-Verkettung. Dies ist aber zu lang, um der Variablen s1 komplett zugewiesen zu werden. Deshalb geht der Rest verloren, und am Bildschirm erscheint nur String-Ver.

In C++ sind die Folgen solcher Fehler viel bedrohlicher: Ein String ist ein Vektor und damit im Prinzip nur ein Zeiger auf den Anfang. Somit gibt es keine Möglichkeit, zu überprüfen, ob der rechtmäßige Bereich eines Strings verlassen wird. Es folgt eine Umsetzung des Pascal-Programms nach C++:

```
#include <iostream.h>
#include <string.h>       // Deklarationen der String-Funktionen
char s1[10], s2[10];

main () {
  strcpy (s1, "String-");
  strcpy (s2, "Verkettung");
  strcat (s1, s2);
  cout << s1;
}
```

Betrachten wir dieses Programm Schritt für Schritt:

Die Deklaration char s1[10] reserviert 10 Bytes. Jetzt ist s1 ein konstanter Zeiger auf char. Das gleiche gilt für s2.

Die Pascal-Anweisung s1 := 'String-' kann man in C++ nicht etwa durch s1 = "String-"; ersetzen. Sie wissen ja bereits, daß man Vektorinhalte nicht einfach zuweisen kann, weil die Variable nur für eine Adresse steht. Beim Zuweisen von Strings verwendet man in C++ die Bibliotheksfunktion strcpy ("string-copy"). Diese Funktion kopiert einfach Byte für Byte vom zweiten String zum ersten, bis sie auf ein Null-Byte stößt.

Mit strcpy (s1, "String-"); werden also die acht Bytes

'S','t','r','i','n','g','-','\0' nach s1 kopiert. In der nächsten Zeile soll der String "Verkettung" nach s2 kopiert werden. Hier geschieht ein Malheur, denn "Verkettung" hat 10 Buchstaben, es werden also mit dem Null-Byte 11 Bytes kopiert, eines mehr als dafür reserviert wurde.

Zum Verketten von Zeichenketten dient die Bibliotheksfunktion strcat. Damit wird der Inhalt des zweiten Arguments dem ersten Argument angehängt

Obwohl die verwendeten Funktionen strcpy und strcat sich in der Standard-Bibliothek befinden, wollen wir hier zur Vertiefung des Verständnisses zumindest die Funktion strcpy nachimplementieren. An diesem Beispiel möchte ich Ihnen auch exemplarisch zeigen, welche Fülle von verschiedenartigen Formulierungen C++ für ein und dieselbe Aufgabe anbietet. Sie sollten versuchen, zumindest passiv die verschiedenen Varianten in ihrer Bedeutung zu erfassen.

Zunächst fassen wir die beiden Strings als Vektoren auf und sprechen die einzelnen Positionen durch Indizes an:

```
enum {false, true};            // Nachbildung des Pascal-Typs boolean
typedef int boolean;

char *strcpy1 (char *Ziel, const char *Quelle) {
  int     i = 0;
  boolean fertig = false;

  while (! fertig) {
    Ziel[i] = Quelle[i];
    if (Quelle[i] == '\0') fertig = true;
    else i = i+1;}
}
```

Byte für Byte wird der Quellstring zum Zielstring kopiert. Wenn das erste Nullbyte kopiert wurde, ist die Arbeit getan. Die boolesche Variable fertig dient hier zum Verlassen der Schleife. Diese Variable läßt sich einsparen, wenn wir eine Endlosschleife verwenden, die mit der Schleifenabbruch-Anweisung break verlassen wird:

```
char *strcpy (char *Ziel, const char *Quelle) {
  int i = 0;
  for (;;) {                        // Endlosschleife
    Ziel[i] = Quelle[i];
    if (Quelle[i] == '\0') break;
    i++;}                           // Kurzschreibweise für i = i+1;
}
```

Eine weitere Vereinfachung ergibt sich, wenn man berücksichtigt, daß der Wert einer Anweisung in C++ gleich dem Wert der rechten Seite ist:

```
char *strcpy (char *Ziel, const char *Quelle) {
  int i = 0;
  while ((Ziel[i] = Quelle[i]) != '\0') i++;
}
```

Ein solch knapper Programmierstil ist sicher eine Zumutung für die meisten Leser.

Trotzdem müssen Sie darauf gefaßt sein, immer wieder solchen Formulierungen von C-Programmierern zu begegnen.

Fast jedes Programm läßt sich noch weiter vereinfachen. Da es in C++ keinen Unterschied zwischen Wahrheitswerten und Zahlen gibt (und da `char` mit `int` verträglich ist), läßt sich jede Abfrage der Form `(Variable != 0)` durch `(Variable)` ersetzen:

```
char *strcpy (char *Ziel, const char *Quelle) {
  int i = 0;
  while (Ziel[i] = Quelle[i]) i++;
}
```

Manche C-Programmierer mögen die `while`-Anweisung nicht und schreiben lieber:

```
char *strcpy (char *Ziel, const char *Quelle) {
  for (int i=0; Ziel[i] = Quelle[i]; i++);
}
```

Der Aufwand für die indizierten Variablen läßt sich einsparen. Es steht ja `s[0]` für `*s`, `s[1]` für `*(s+1)` usw. Anstatt jeweils den Index zu erhöhen, können wir auch die Zeiger selbst erhöhen:

```
char *strcpy (char *Ziel, const char *Quelle) {
  while (*Ziel = *Quelle) {Quelle++; Ziel++;}
}
```

Im Standardwerk von Kernighan/Ritchie [Kern88] wird folgende Extrem-Lösung vorgeschlagen:

```
char *strcpy (char *Ziel, const char *Quelle) {
  while (*Ziel++ = *Quelle++);
}
```

Hier wird in der `while`-Bedingung das Kopieren und das Erhöhen der Zeiger verpackt. Zum Verständnis müssen Sie wissen, daß unäre Operatoren rechtsassoziativ sind. `*Ziel++` steht also für `*(Ziel++)` und nicht etwa für `(*Ziel)++`. Weiter müssen Sie sich klarmachen, daß der Wert von `Ziel++` die Adresse vor dem Erhöhen ist. Ein solch kryptischer Stil wird immerhin von modernen Compilern mit mehreren Warnungen bedacht: Die Form `while (a = b)` könnte ein Pascal-Programmierer versehentlich statt `while (a == b)` geschrieben haben. Außerdem ist eine leere Anweisung nach einer `while`-Bedingung verdächtig. In diesem Fall kann man diese Warnungen ignorieren, aber im allgemeinen können Sie sich viel Kummer ersparen, wenn Sie jede Warnung des Compilers überprüfen.

Schließlich sehen Sie das Beispiel noch einmal für `for`-Liebhaber:

```
char *strcpy (char *Ziel, const char *Quelle) {
  for (; *Ziel++ = *Quelle++;);
}
```

Übungsaufgabe 6:

Was bewirkt das folgende Programmstück?

```
char s[10];
strcpy (s, "Vorsicht!");
strcpy (s+1, s);
```

Übungsaufgabe 7:

Testen Sie folgendes Programm und erklären Sie das merkwürdige Verhalten:

```
#include <iostream.h>
#include <string.h>

main () {
  char s1[6], s2[4], s3[6];
  strcpy (s3, "BAR");
  strcpy (s2, "WIRKLICH");
  strcpy (s1, "WUNDER");
  strcat (s1, s3);
  cout << "\n" << s1;
}
```

Zeiger auf Funktionen

In C++ können Sie keine Variablen erklären, die Funktionen als Wert haben. Dies scheitert technisch daran, daß je zwei Funktionen im allgemeinen verschiedenen Speicherbedarf haben. Erlaubt sind dagegen Zeiger auf Funktionen. Ebenso wie Datenzeiger sind Funktionszeiger typisiert. Zwei Funktionszeiger sind nur dann kompatibel, wenn sie auf Funktionen mit dem gleichen Parameterschema (gleiche Anzahl, gleiche Typen) und typgleichem Funktionswert zeigen.

Wie werden Zeiger auf Funktionen definiert? Nehmen Sie eine Funktionsdeklaration in der Kurzform ohne Parameternamen, setzen Sie vor den Funktionsnamen einen Stern und klammern Sie den Funktionsnamen mit dem Stern ein! Zum Beispiel wird mit

```
char Summe (int, int);
```

eine Funktion deklariert, die zwei Argumente vom Typ `int` verlangt und einen Wert vom Typ `char` zurückgibt. Mit

```
char (*FktZeiger) (int, int);
```

wird ein Zeiger auf eine solche Funktion definiert. Die Klammern um den Namen mit dem Stern dürfen Sie nicht vergessen, denn mit

```
char *Zeiger (int, int);
```

wird kein Zeiger auf eine Funktion deklariert, sondern eine Funktion, die einen Zeiger auf `char` zurückgibt.

Wenn Sie einem Zeiger auf eine Funktion etwas zuweisen möchten, müßten Sie eigentlich

die Adresse einer Funktion bilden. Um die Schreibweise zu vereinfachen, gilt hier eine ähnliche Konvention wie bei den Vektortypen: Funktionsnamen stehen prinzipiell für einen Zeiger auf die Funktion. Erst durch die runden Klammern der Parameter dahinter wird der Zeiger dereferenziert und die Funktion aufgerufen. Deshalb müssen in C++ auch beim Aufruf einer Funktion ohne Parameter die runden Klammern stehen!

Das folgende Beispiel zeigt, wie Zeiger auf Funktionen definiert werden, und wie man ihnen Funktionen als Wert zuweist:

```
int (*f) (int);     // f ist Zeiger auf Funktion mit einem Argument
                    // vom Typ int und Ergebnistyp int
int f1 (int i)   {return i;}
int f2 (int n)   {return 2*n;}
int f3 (float x) {return 3*x;}

main () {
  f = f1;           // steht für f = &f1;
  n = f(1);         // steht für n = (*f)(1); jetzt ist n = 1
  f = f2;
  n = f(1);         // jetzt ist n = 2
  f = f3;           // Fehler: unverträgliche Zeigertypen!
}
```

Das folgende Beispiel eines einfachen Sortieralgorithmus (*Bubblesort*) zeigt eine sinnvolle Anwendung von Funktionszeigern. Die Funktion `bsort` kann Vektoren eines beliebigen Datentyps sortieren. Dazu wird die Adresse des Vektors, die Anzahl der Elemente und die Größe des Datentyps in Bytes übergeben. Zum Sortieren muß außerdem noch gesagt werden, wie zwei Elemente verglichen werden, denn das hängt von der Interpretation der Daten ab. So kann zum Beispiel ein Speicherwort (2 Bytes), in dem alle Bits gesetzt sind, die `unsigned`-Zahl 65535 oder die `int`-Zahl -1 bedeuten.

```
// BSORT.CPP
#include <iostream.h>

void vertausche (void *p1, void *p2, size_t Groesse) {
  char h;
  char *cp1 = p1, *cp2 = p2;
  for (int i=0; i<Groesse; i++) {
    h = cp1[i];
    cp1[i] = cp2[i];
    cp2[i] = h;}
}

void bsort (void  *Basis,       // Adresse des Sortierbereichs
            size_t Anzahl,      // Anzahl der Elemente
            size_t Groesse,     // Größe eines Elements
            int (*Vergleichsfunktion)(const void*, const void*)) {
  char *i, *j;
  for (i=Basis+(Anzahl-1)*Groesse; i>Basis; i-=Groesse)
    for (j=Basis; j<i; j+=Groesse)
      if ((*Vergleichsfunktion) (i, j) < 0)
        vertausche (i, j, Groesse);
}

int VglInt (int *i, int *j) {return *i - *j;}

main () {
  int i;
  static int A[10] = {31, 41, 59, 26, 53, 58, 97, 93, 23, 84};
  cout << "\nunsortiert:\n";
  for (i=0; i<10; i++) cout << A[i] << " ";
```

```
  bsort (A, 10, sizeof (int), VglInt);
  cout << "\nsortiert:\n";
  for (i=0; i<10; i++) cout << A[i] << " ";
  cout << "\n";
}
```

Bitte bedenken Sie, daß der Bubblesort-Algorithmus bei größeren Datenmengen ziemlich ineffektiv arbeitet. In der Standard-Bibliothek von C (`STDLIB.H`) gibt es auch eine Quicksort-Implementierung `qsort`, die mit genau den gleichen Parametern aufgerufen wird.

1.11 Deklarationen und Definitionen

Die beiden Begriffe *Deklaration* und *Definition* sind geeignet, vollständige Verwirrung hervorzurufen, weil ihre Bedeutung in verschiedenen Kontexten völlig unterschiedlich gehandhabt wird.

Folgen wir einem modernen Standardwerk der Informatik [Watt90], so ist eine Deklaration ein Programmstück, das einen Namen mit einem Objekt im weitesten Sinne verbindet. Existiert dieses Objekt bereits, so spricht man von einer Definition. Einige Beispiele in Pascal mögen dies klären:

```
const n = 1;         (* Definition: Bindung des Namens n an Zahl *)
type Zahl = integer;(* Deklaration: Zahl ist neuer Typ! *)
var i: integer;      (* Deklaration: neue Variable wird erzeugt *)
```

In der Pascal-Terminologie spricht man gewöhnlich nur von Deklarationen. Durch eine Deklaration wird im Normalfall eine Variable eingeführt oder eine Funktion (oder Prozedur) definiert. Im Gegensatz dazu wird durch eine `forward`-Deklaration keine Funktion definiert, sondern nur die Schnittstelle einer an anderer Stelle definierten Funktion erklärt.

Die Standardwerke über C++ sind sich darin einig, daß eine Deklaration, die die angesprochene Variable oder Funktion wirklich bereitstellt, *Definition* genannt wird. Der Begriff *Deklaration* wird nicht einheitlich verwendet. Bei Stroustrup [Stro86] wird er als Oberbegriff benutzt. Es fehlt also ein Begriff für eine "Nur-Deklaration". Lippman [Lipm89] versteht unter einer Deklaration eine "Nur-Deklaration". In diesem Fall fehlt der Oberbegriff. Um nicht ein neues Wort schöpfen zu müssen, wollen wir uns im allgemeinen der Sprechweise von Stroustrup anschließen, aber Ausnahmen machen, wenn dies aus dem Zusammenhang klar wird.

In Pascal besteht eine klare Trennung zwischen Deklarationen und Anweisungen: Jeder Block besteht aus einem Deklarationsteil und einem Anweisungsteil. Im Deklarationsteil gibt es wiederum einen speziellen Abschnitt für Variablendeklarationen (beginnend mit dem Schlüsselwort `var`). Turbo Pascal hat zwar die Reihenfolge innerhalb des Deklarationsteils liberalisiert, die Trennung zwischen Deklarations- und Anweisungsteil bleibt aber bestehen. In C++ gibt es diese Trennung nicht. Hier dürfen Deklarationen fast überall stehen. Wir wollen auf die Möglichkeiten im einzelnen eingehen:

Variablen

Eine *Variablendefinition* besteht aus der Typbezeichnung, gefolgt vom Namen. Die Variable kann bei der Definition schon mit einem *Anfangswert* initialisiert werden.

```
int n;            // Einfache Definition
int a = 1;        // Initialisierung
int i, j = 3, k;  // mehrere Variable gleichen Typs
                  // können gemeinsam definiert werden
```

Globale Variablen sind alle Variablen, die außerhalb einer Funktionsdefinition definiert oder deklariert werden. Eine globale Variable lebt während der ganzen Laufzeit des Programms. Sie muß in genau einer Datei einmal definiert werden. Wenn sie in anderen Dateien angesprochen werden soll, muß sie dort deklariert werden. Eine Deklaration wird durch das vorangestellte Schlüsselwort `extern` gekennzeichnet. Im folgenden Beispiel wird in der Datei 1 eine Variable `n` vom Typ `int` definiert. Diese Variable kann zunächst nur in der Datei angesprochen werden, in der sie definiert wurde. Soll sie in einer anderen Datei angesprochen werden, muß sie dort zunächst deklariert werden.

```
// Datei 1:
int n;         // hier wird die Variable n definiert
...

// Datei 2:
extern int n;  // Deklaration der anderswo definierten Variablen n
```

Bei einer Deklaration stellt der Compiler keinen Speicherplatz zur Verfügung, sondern erhält nur die Information, daß anderswo eine solche Variable existiert. Wenn Sie versehentlich eine Variable deklarieren, die nirgendwo definiert ist, kann dieser Fehler erst vom Binder erkannt werden.

Das Schlüsselwort `extern` suggeriert leicht ein Mißverständnis. Das mit `extern` deklarierte Objekt muß nicht in einer anderen Datei definiert werden. Es kann auch in der gleichen Datei an anderer Stelle definiert sein. Mit `extern` wird lediglich eine Deklaration von einer Definition unterschieden.

Manchmal ist es angebracht, die *Sichtbarkeit* einer globalen Variablen auf die Datei zu beschränken, in der sie definiert ist. Hierzu dient das Schlüsselwort `static`.

```
static int n; // n ist aus anderen Dateien nicht ansprechbar
```

Lokale Variablen werden innerhalb eines Blocks (also auch innerhalb einer Funktion) definiert. Sie sind nur in dem Block sichtbar, in dem sie definiert sind. Sie können an beliebiger Stelle im Block definiert werden, dürfen aber nicht vor ihrer Definition angesprochen werden. Lokale Variablen können nicht deklariert werden. Dies wäre auch nicht sinnvoll, da sie außerhalb des Blocks nicht sichtbar sind.

In Pascal werden alle lokalen Variablen bei ihrer Definition (auf dem Stack) angelegt und beim Verlassen des Blocks wieder beseitigt. Dadurch wird es oft notwendig, globale

Variablen anzulegen, die nur für eine bestimmte Funktion relevant sind. In C++ können Sie auch lokale Variablen definieren, deren Lebensdauer die Programmlaufzeit ist. Hierzu wird ebenfalls das Schlüsselwort `static` verwendet. Damit ist es möglich, Funktionen mit einem "Gedächtnis" auszustatten. Betrachten Sie als Beispiel folgenden Zufallszahlengenerator:

```
int random (int n) { // erzeugt Zufallszahl im Bereich 0 ... n-1
  static unsigned long r = 1;
  r = r * 1103515245 + 12345;
  return (unsigned) (r >> 16) % n;
}
```

Gegenüber einer globalen Variablen, wie es in Pascal erforderlich wäre, bietet die lokale `static`-Variable zwei Vorteile: Erstens wird so vermieden, daß inhaltlich zusammengehörende Informationen im Programm verstreut werden müssen. Zweitens ist die lokale Variable vor versehentlichem Zugriff von außerhalb der Funktion geschützt.

Wenn eine globale und eine lokale Variable den gleichen Namen haben, kann in Pascal nur die lokale Variable angesprochen werden. In C++ können mit dem unären Präfix-Operator `::` globale Variablen auf Programmebene angesprochen werden, wenn sie mit einer lokalen Variablen kollidieren. Im folgenden Beispiel werden drei Variablen `n` definiert: Auf Programmebene (mit Wert 1), im Hauptblock der Funktion `main` (Wert 2) und in einem inneren Block der Funktion `main` (Wert 3). Vom inneren Block aus ist die Variable des umfassenden Blocks nicht ansprechbar!

```
// Test des Global-Operators ::

#include <stream.h>
int n = 1;

main () {
  int n = 2;
  {
    int n = 3;
    cout << "lokal: " << n << ", global: " << ::n << "\n"; // 3, 1
  }
  cout << "lokal: " << n << ", global: " << ::n << "\n"; // 2, 1
}
```

Funktionen

Sie wissen bereits, wie Funktionen definiert werden. Für eine *Funktionsdeklaration* nimmt man die Funktionsdefinition ohne den Funktionsblock, setzt ein Semikolon dahinter und das Schlüsselwort `extern` davor.

```
extern int Summe (int a, int b); // Deklaration der Funktion Summe
...
int Summe (int a, int b) {       // Definition der Funktion Summe
  return a + b;
}
```

Das Schlüsselwort `extern` darf bei der Deklaration auch weggelassen werden.

Ebenso wie bei Variablen kann die *Sichtbarkeit* einer Funktion mit dem Schlüsselwort

static auf die Datei beschränkt werden, in der sie definiert ist.

Funktionsdeklarationen können zum einen wie die forward-Deklarationen in Pascal verwendet werden, etwa bei sich wechselseitig aufrufenden rekursiven Funktionen. Zum anderen werden sie benötigt, um Funktionen verwenden zu können, die in einer anderen Datei definiert sind.

Übersicht: Sichtbarkeit und Lebensdauer

	globale Variablen, Funktionen	**lokale Variablen**
Definition	außerhalb von Funktionen	innerhalb von Funktionen
Deklaration	mit extern: außerhalb oder	keine weitere Deklaration
	innerhalb von Funktionen	
Sichtbarkeit	mit static: Datei	Block
	sonst: alle Dateien	
Lebensdauer	Programmlaufzeit	mit static: Programmlaufzeit
		sonst: Block-Ausführung

Komplexe Deklarationen

Verschachtelte Deklarationen sind in C++ nicht ganz einfach zu durchschauen, weil der Aufbau nicht, wie in Pascal, geradlinig von links nach rechts geschieht, sondern unter Berücksichtigung der Vorrangregeln. Eine Deklaration in C++ besteht aus einem Typnamen und dahinter einem verschachtelten Ausdruck, der den Namen des zu deklarierenden Objekts enthält. Zum Verständnis muß man wissen, daß dieser Ausdruck ein Objekt vom Typ des davorstehenden Typnamens bezeichnet. Ein Beispiel soll dies verdeutlichen:

```
int (*a)[n];
```

Diese Deklaration definiert ein Objekt a mit der Eigenschaft, daß (*a)[n] vom Typ int ist. Welchen Typ hat demnach a selbst? Die Frage läßt sich schrittweise beantworten:

(*a)[n] ist von Typ int.

(*a) ist vom Typ Vektor von int.

a ist vom Typ Zeiger auf Vektor von int.

Wichtig für die richtige Interpretation verschachtelter Deklarationen ist die Kenntnis der Vorrangregel: Unäre (einstellige) Operatoren - und dazu gehören insbesondere die Vektorklammern [] und die Funktionsklammern () - sind *rechtsassoziativ*. Um es nochmals zu wiederholen: Wenn rechts und links von einem Ausdruck ein solcher Operator steht, wird zunächst der rechte Operator angewendet. Lassen wir in der obigen Deklaration die runden Klammern weg, ändert sich die Bedeutung:

```
int *a[n]; // vollständig geklammert: int *(a[n]);
```

`*(a[n])` ist von Typ `int`.

`a[n]` ist vom Typ Zeiger auf `int`.

a ist vom Typ Vektor von Zeigern auf `int`.

Statt dieser schrittweisen Zerlegung von außen nach innen lassen sich komplexe Deklarationen mit etwas Übung unmittelbar von innen (beim zu deklarierenden Namen beginnend) nach außen (beim Typnamen endend) lesen, indem man gemäß der Vorrangregel, wenn möglich, zunächst nach rechts geht:

```
int *(a[n]);
 3  2 0 1
```

Die Zahlen 0 bis 3 zeigen hier die Schritte: (0) a ist (1) Vektor von (2) Zeigern auf (3) `int`.

Machen Sie sich bitte den Unterschied zwischen den beiden folgenden Deklarationen klar:

```
int (* f)()
 3   1 0 2     (0) f ist (1) Zeiger auf (2) Funktion mit Wert (3) int.

int * f ()
 3  2 0 1      (0) f ist (1) Funktion mit Wert (2) Zeiger auf (3) int.
```

Ein letztes Beispiel dient als nicht nachahmenswerter Härtetest. Im nächsten Abschnitt sehen Sie, wie man komplexe Deklarationen vermeiden kann.

```
int * ( * ( *  f ) () ) []
 6  5   3   1  0   2    4
```

(0) `f` ist (1) Zeiger auf (2) Funktion mit Wert (3) Zeiger auf (4) Vektor von (5) Zeigern auf (6) `int`.

Übungsaufgabe 8:

Orden Sie die Variablen `A` ... `H` im folgenden Pascal-Programm den entsprechenden

Variablen a ... h im anschließenden C++ Programm zu:

```
program PascalProgramm;
const n=9;
type
  ArrayOf_Int = array[0..n-1] of integer;
  ArrayOf_ArrayOf_Int = array[0..n-1] of ArrayOf_Int;
  PointerTo_Int = ^integer;
  PointerTo_ArrayOf_Int = ^ArrayOf_Int;
  ArrayOf_PointerTo_Int = array[0..n-1] of PointerTo_Int;
  PointerTo_PointerTo_Int = ^PointerTo_Int;
var
  A: array[0..n] of array[0..n-1] of array[0..n-1] of integer;
  B: array[0..n] of array[0..n-1] of ^integer;
  C: array[0..n] of ^ArrayOf_Int;
  D: ^ArrayOf_ArrayOf_Int;
  E: array[0..n] of ^PointerTo_Int;
  F: ^ArrayOf_PointerTo_Int;
  G: ^PointerTo_ArrayOf_Int;
  H: ^PointerTo_PointerTo_Int;
begin
end.
```

```
// C++ Programm
const int n=10;
int ***a, *(*b)[n], **c[n], (**d)[n],
    e[n][n][n], (*f)[n][n], *g[n][n],
    (*h[n])[n];
main () {}
```

Typbezeichnungen

Von Pascal kennen Sie die Möglichkeit, eigene Typen zu definieren. Dabei ist zu beachten, daß jede Typdefinition einen eigenen Typ definiert. Betrachten wir ein Beispiel:

```
type
  typ1 = ^integer;
  typ2 = ^integer;
var
  a,b: typ1;
  c  : typ2;
  d  : ^integer;
begin
  a := b;  (* korrekt: a und b vom gleichen Typ *)
  a := c;  (* Type mismatch *)
  a := d;  (* Type mismatch *)
end.
```

Sie sehen, daß zwei Typen auch dann inkompatibel sind, wenn sie identisch definiert sind. In C++ ist die Situation etwas verworren. Vergegenwärtigen Sie sich bitte, daß es in C++ zwei unterschiedliche Methoden zur Bildung abgeleiteter Datentypen gibt: Zum einen gibt es die Operatoren `*`, `[]` und `()` zur Bildung von Zeiger-, Vektor- und Funktionstypen, zum anderen die Schlüsselwörter `struct` und `union` zur Bildung von Strukturen und Varianten. Zwischen diesen beiden Gruppen besteht auch ein wesentlicher formaler Unterschied. Mit den Operatoren werden namenlose Typen gebildet, während die beiden Schlüsselwörter neue Typen im Sinne von Pascal bilden.

Um den namenlosen Typen auch Bezeichnungen geben zu können, hat C++ die `typedef`-

Anweisung von Standard-C übernommen. Diese gleicht einer Variablendefinition mit dem vorangestellten Schlüsselwort typedef. Hiermit wird anstelle eines Variablennamens eine Typbezeichnung eingeführt. So wird etwa mit

```
typedef int *intZeiger;
```

eine Bezeichnung für den Datentyp Zeiger auf int eingeführt.

Mit der typedef-Anweisung wird nur ein alternativer Name (Synonym) für einen Typ eingeführt, jedoch kein neuer Typ definiert.

Betrachten Sie die feinen Unterschiede im folgenden Beispiel:

```
struct typ1 {int x, y;}; // typ1 und typ2 sind strukturgleich,
struct typ2 {int x, y;}; // aber nicht verträglich!

typedef typ2 typ3;       // typ3 ist Synonym zu typ2

typedef int *intZeiger;

typ1      a;
typ2      b;
typ3      c;
intZeiger p;
int       *q;

main () {
  a = b;  // Fehler: typ1 und typ2 sind verschiedene Datentypen
  b = c;  // korrekt: typ2 und typ3 sind Synonyme
  p = q;  // korrekt: intZeiger ist Synonym für int *
}
```

Übungsaufgabe 9:

Welcher Datentyp wird durch folgende Deklaration eingeführt?

```
int (*(* A[n])(int))[n];
```

Definieren Sie diesen Typ mit einer Folge von einfachen Typdeklarationen!

2 Projektentwicklung in C++

Nachdem Sie nun die Sprachmittel für die konventionelle Programmierung in C++ kennengelernt haben, befassen wir uns in diesem Kapitel mit dem Prozeß der Programmentwicklung und den dazu nötigen Hilfsmitteln.

2.1 Compilersteuerung mit dem Präprozessor

Wir haben schon in vielen Beispielen die Präprozessor-Direktiven #include und #define verwendet. Hier finden Sie eine Zusammenstellung, die auch weitere Möglichkeiten des

Präprozessors umfaßt.

Vor jedem Compilerlauf wird die Quelldatei mit dem Präprozessor bearbeitet. Dabei werden Kommentare ausgeblendet und Präprozessor-Direktiven ausgeführt. Der Compiler von Borland C++ umfaßt den Präprozessor und erzeugt keine Zwischendatei. Es kann aber manchmal nützlich sein, die Wirkung des Präprozessors zu sehen, vor allem, um unerklärliche Fehler aufzuspüren. Deshalb gibt es bei Borland C++ den Präprozessor auch als eigenständiges Programm.

Präprozessor-Direktiven sind vom eigentlichen Programmtext durch das Zeichen # am Zeilenanfang klar zu unterscheiden. Im Gegensatz zur Syntax von C++, für die Zeilenwechsel keine Bedeutung haben, muß jede Compiler-Direktive mit einem Zeilenwechsel abgeschlossen werden.

Einfügen von Dateien

Eine Direktive der Form

`#include "`**Dateiname**`"`

veranlaßt, daß diese Zeile durch den Inhalt der Datei **Dateiname** ersetzt wird. Die Datei wird zunächst im aktuellen Verzeichnis gesucht, anschließend werden die voreingestellten Include-Verzeichnisse nach der Datei durchsucht. Wird der Dateiname statt in Anführungszeichen in spitze Klammern eingeschlossen, werden nur die voreingestellten Include-Verzeichnisse nach der Datei durchsucht.

```
#include <stdlib.h>    // im Include-Verzeichnis
#include "hilfe.h"     // im aktuellen Verzeichnis
```

Wie die Include-Verzeichnisse spezifiziert werden, ist von der Implementierung des Compilers abhängig. Bei den meisten C-Compilern wird dafür eine Umgebungsvariable des Betriebssystems gesetzt. In der integrierten Umgebung von Borland C++ werden die Verzeichnisse im Menü (*Options - Directories*) angegeben.

`#include`-Direktiven dürfen verschachtelt werden: Eingebundene Dateien dürfen wieder `#include`-Direktiven enthalten.

Textersetzungen

Die `#define`-Direktive erlaubt *Textersetzungen* (*Makros*) mit oder ohne Parameter. Sie sind ein wichtiges Hilfsmittel in Standard-C. Sie haben aber auch einen großen Nachteil: Mit Textersetzungen läßt sich sehr leicht die Typstrenge von C++ unterlaufen. In C++ sollten Konstanten-Definitionen, Aufzählungen oder inline-Funktionen bevorzugt werden. Sie dürfen den kurz gehaltenen Rest dieses Abschnitts deshalb mit gutem Gewissen überfliegen.

Ein Makro ohne Parameter hat folgende Form:

`#define` **Name Ersetzungstext**

Dabei muß der Name ein gültiger Bezeichner sein, für den Ersetzungtext gibt es keine Einschränkungen. Er kann sogar leer sein. Hinter dem Namen darf auch eine Parameterliste stehen (Argumente: gültige Bezeichner, durch Kommata getrennt, in runde Klammern eingeschlossen). Zwischen dem Namen und der öffnenden Klammer der Parameterliste darf kein Lückenzeichen (Leerzeichen, Tabulator etc.) stehen.

Definierte Makros können mit der `#undef`-Direktive wieder aufgehoben werden.

Übungsaufgabe 10:

Untersuchen Sie das folgende Programm. Erklären Sie die Fehler und ersetzen Sie die Makros durch `inline`-Funktionen:

```
#define Quadrat(x) x*x
#define Doppelt (x) x+x

main () {
  int a,b,c;
  a = Quadrat (b+c);
  a = Doppelt (b+c);
}
```

Bedingte Compilierung

Oft ist es nötig, mehrere Versionen eines Programms zu entwickeln, die sich nur in wenigen Details unterscheiden. Diese Unterschiede können in der Anpassung an eine bestimmte Hardware-Konfiguration (Grafik-Auflösung, Koprozessor etc.) oder an verschiedene Compiler begründet sein. Man könnte hierzu eine Kopie des Quelltextes anfertigen und darin die Änderungen anbringen. Oft fallen aber später Korrekturen im versionsunabhängigen Programmteil an. Diese müssen dann immer in allen Versionen angebracht werden. Dabei geht leicht die Übersicht verloren.

Man kann sich viel Verwirrung und Speicheraufwand ersparen, indem man die verschiedenen Versionen mit den Compiler-Direktiven zur bedingten Compilierung verwaltet:

```
#ifdef Bezeichner
Programmstück1
#else
Programmstück2
#endif
```

Das **Programmstück1** wird übersetzt, falls der Bezeichner (mit `#define`) definiert ist. Ist er nicht definiert, so wird das **Programmstück2** übersetzt. Die Definition des Bezeichners sollte ganz am Anfang des Programms stehen. So läßt sich durch Ändern einer Zeile zwischen zwei Programmversionen umschalten. Wie üblich, darf der `else`-Zweig auch

entfallen (#else-Direktive und Programmstück2). Mit der #ifndef-Direktive kann auch geprüft werden, ob der Bezeichner nicht definiert ist. Die Direktiven zur bedingten Compilierung dürfen selbstverständlich verschachtelt werden.

In der integrierten Umgebung von Borland C++ können #define-Anweisungen auch im Menü *Options - Compiler - Code Generation - Defines* angegeben werden. Das ist vor allem dann praktisch, wenn diese Definitionen in mehreren Dateien benötigt werden.

2.2 Modularisierung

Bisher haben wir nur Programme betrachtet, deren Quelltext in einer Datei abgespeichert ist. Nicht ganz kleine Programmierprojekte werden aber in der Praxis sinnvollerweise modular aufgebaut: Der Programmtext wird so in mehrere Dateien zerlegt, daß einzelne dieser Dateien möglichst für andere Zwecke wiederverwendet werden können. Jede dieser Dateien wird einzeln übersetzt und die übersetzten Objekte zu einem lauffähigen Programm zusammengebunden. Turbo Pascal bietet hierfür (ab der Version 4.0) das *Unit-Konzept* an.

Eine erste Vorstellung, wie Sie in C++ Programme modular aufbauen können, sollen Ihnen die folgenden Programmskelette geben:

```
unit Hilf;                             // hilf.cpp
interface
  var i: integer;                      int i;
  function f (i: integer): char;
implementation
  var k: integer;                      static int k;
  procedure p (i: integer);            static void p (int i) {
    begin
      ...                                ...
    end;                               }

  function f (i: integer): char;       char f (int i) {
    begin
      ...                                ...
    end;                               }
  end.

program Haupt;                         // haupt.cpp
uses Hilf;                             extern int i;
                                       extern char f (int i);
var k: integer;                        static int k;

begin                                  void main () {
  ...                                    ...
end.                                   }
```

Beachten Sie folgenden Unterschied: Zu einem Pascal-Programm gehören genau eine program-Datei und beliebig viele unit-Dateien. In C++ gibt es keinen formalen Unterschied zwischen den einzelnen Dateien. Alle können Deklarationen und Funktionen in beliebiger Folge enthalten, aber genau eine Datei muß die Hauptfunktion main enthalten.

2.3 Header-Dateien

Die mit einem Modul benötigten Deklarationen faßt man üblicherweise in einer *Header-Datei* zusammen. Diese erhält die Extension `.h`. Damit sieht die Gegenüberstellung der Hauptmoduln so aus:

```
program Haupt;
uses Hilf;

var k: integer;

begin
  ...
end.
```

```
// haupt.cpp
#include <hilf.h>

static int k;

void main () {
  ...
}
```

Die Header-Datei könnte so aussehen:

```
// hilf.h
extern int i;
extern char f (i: integer);
```

Innerhalb von Include-Dateien können wiederum Dateien mit `#include` eingebunden werden.

```
// a.h
#include <b.h>
#include <c.h>
...

// b.h
#include <d.h>
...


// c.h
#include <d.h>
...
```

Mit dem Präprozessorbefehl `#include <a.h>` wird die Datei `d.h` zweimal eingebunden. Um dies zu verhindern, sollte man alle Header-Dateien nach folgendem Schema absichern:

```
// hilf.h
#ifndef HILF_H
#define HILF_H
extern int i;
char f (i: integer): char;
#endif
```

Beim ersten Einbinden dieser Datei ist dem Präprozessor der Name `HILF_H` unbekannt. Deshalb wird der Text zwischen `#ifndef` und `#endif` eingelesen. Hierbei wird der Name `HILF_H` definiert (als leere Zeichenkette) und ist dem Präprozessor beim zweiten Einbinden dieser Datei bekannt. In diesem Fall wird der Text zwischen `#ifndef` und `#endif` übersprungen.

2.4 Verwaltung von Programm-Projekten

Wenn Sie ein Programm in einzelne Moduln zerlegen, können Sie nicht nur die Überschaubarkeit erhöhen, Sie sparen auch noch wertvolle Entwicklungszeit ein. Denn nach einer Änderung in einem der Programm-Moduln brauchen Sie nicht Ihr ganzes Programm neu zu übersetzen, sondern nur den veränderten Modul.

Bei größeren Projekten ist es aber nicht ganz einfach, die notwendigen Konsequenzen einer Änderung zu überschauen. Haben Sie etwa Korrekturen in einer Header-Datei angebracht, so müssen alle Quelltexte neu übersetzt werden, die diese Datei (eventuell indirekt) importieren. Anschließend muß der Linker aufgerufen werden, um das ausführbare Programm auf den neuesten Stand zu bringen.

Eine unbefriedigende Lösung des Problems wäre eine Stapeldatei, die pauschal sämtliche Schritte der Übersetzung und Bindung erledigt. Damit ginge aber der zeitliche Vorteil der getrennten Übersetzung verloren. Im Betriebssystem UNIX gibt es deshalb das Dienstprogramm MAKE, das automatisch die erforderlichen Schritte ausführt. Dazu müssen die Abhängigkeiten der Datei zunächst einmal in einer Projektbeschreibungsdatei (MAKEFILE) dokumentiert werden. Es folgt ein Beispiel (für den Zortech-Compiler), für ein Programm, das von zwei Quelldateien und einer Header-Datei abhängt.

```
grafdemo.exe: grafdemo.obj grafik.obj
    ztc -ml -ografdemo grafdemo.obj grafik.obj

grafdemo.obj: grafik.h grafdemo.cpp
    ztc -ml -c -ografdemo.obj grafdemo.cpp

grafik.obj: grafik.h grafik.cpp
    ztc -ml -c -ografik.obj grafik.cpp
```

Diese Datei besteht aus drei Regeln. Jede Regel besteht aus zwei Zeilen. In der ersten Zeile steht der Name des gewünschten Produkts, dann ein Doppelpunkt und danach die benötigten "Zutaten". In der zweiten Zeile (die eingerückt sein muß) folgt das "Rezept" zur Herstellung des Produkts.

Das Dienstprogramm MAKE prüft, ob eine oder mehrere der Zutaten neueren Datums als das Produkt sind. Nur in diesem Fall wird das Rezept neu ausgeführt. Wurde in unserem Beispiel die Datei `grafdemo.cpp` geändert, so betrifft dies zunächst die zweite Regel: `grafdemo.cpp` wird neu übersetzt. Danach ist `grafdemo.obj` neuer als `grafdemo.exe`. Es wird also auch die erste Regel ausgeführt. Danach ist das Projekt auf dem neuesten Stand, ohne daß `grafik.cpp` neu übersetzt wurde.

Dieses einfache Beispiel konnte natürlich nur ein Schlaglicht auf MAKE werfen. Tatsächlich gibt es noch weitere Möglichkeiten, wie implizite Regeln und Makros.

Wenn Sie in Borland C++ programmieren, steht Ihnen das MAKE-Programm zwar zur Verfügung, Sie benötigen es aber nur, wenn Sie die Kommandozeilenversion des Compilers verwenden. Innerhalb der integrierten Oberfläche von C++ können Sie die eingebaute *automatische Projektverwaltung* verwenden. Sie erklärt sich fast selbst. Sie brauchen nur

in einem Dialog die Programmdateien anzugeben, aus denen Ihr Projekt besteht. Die Abhängigkeit von den darin importierten Header-Dateien findet die Projektverwaltung selbst heraus!

2.5 Bibliotheken

Wenn Sie Programmdateien geschrieben haben, die Sie immer wieder in verschiedenen Projekten benötigen, und die soweit ausgereift sind, daß sie nicht mehr ständig geändert werden müssen, gibt es einen bequemeren Weg als die Dateien mit Hilfe der Projektverwaltung einzubinden. Sie können mehrere Objekt-Dateien mit Hilfe des Bibliotheksverwaltungsprogramms (bei Borland C++ heißt es TLIB.EXE) in eine Bibliothek zusammenfassen. Entweder erzeugen Sie eine neue Bibliothek oder fügen die Dateien in die Standard-Bibliothek ein. Die Bedienung von TLIB ist im Benutzerhandbuch von Borland C++ (bei Turbo C++ in einer mitgelieferten Datei) genau beschrieben.

Falls Sie daran interessiert sind, Programme zu erzeugen, die bescheiden mit dem Speicherplatz umgehen, sollten Sie folgendes beachten: Während Turbo Pascal beim Binden optimiert und nur die Prozeduren einbindet, die auch wirklich verwendet werden, werden bei C++ immer komplette Objekt-Dateien eingebunden. Deshalb kann es sich auszahlen, wenn Sie anstelle einer großen Objektdatei eine Bibliothek mit vielen kleinen Objektdateien verwenden.

2.6 Bibliotheksfunktionen

Zur Sprache C gehört nach der ANSI-Norm eine reichhaltige Funktionsbibliothek. Dies ermöglicht eine weitgehende Portabilität von C-Programmen, denn man kann sich darauf verlassen, daß diese Funktionen in jeder neueren Implementierung vorhanden sind.

Hier finden Sie nur eine Auswahl wichtiger Funktionen. Zunächst werden die Funktionen zur Datei-Behandlung von ANSI-C dargestellt. Sie sind in der Datei `STDIO.H` deklariert. Anschließend finden Sie kurze Hinweise zu weiteren wichtigen Standard-Funktionen.

In C++ gibt es auch eine objektorientierte Ein-/Ausgabe-Bibliothek (Header-Datei: `IOSTREAM.H`), die allerdings noch nicht genormt ist. In diesem Buch verwenden wir nur elementare Möglichkeiten der objektorientierten Ein-/Ausgabe. Über weitere Details und über weitere Funktionen können Sie sich jederzeit mit der eingebauten Hilfe-Funktion von Borland C++ informieren (Cursor auf den Funktionsnamen setzen und Strg-F1 drücken).

Bildschirmausgabe

```
printf (const char *format, ...);
```

Die Funktion `printf` (Abkürzung für "print formatted") dient zur formatierten Ausgabe

am Bildschirm und bietet eine Fülle möglicher Darstellungsarten, für deren Details Sie bitte Ihr Referenzhandbuch oder die eingebaute Hilfe von Borland C++ zu Rate ziehen. Zunächst fallen die drei Punkte am Ende der Deklaration auf. Damit wird dem Compiler mitgeteilt, daß die Funktion eine variable Anzahl von Argumenten akzeptiert. Nur der Typ der ersten Arguments wird geprüft. Das erste Argument ist der *Format-String*. Er kann beliebig viele *Konvertierungs-Anweisungen* enthalten, die jeweils mit einem Prozent-Zeichen beginnen. Zu jeder Konvertierungsanweisung im Format-String muß anschließend genau ein Argument des entsprechenden Typs folgen.

```
int puts (const char *str);
```

`puts` (Abkürzung für "put string") gibt den mit einem Nullbyte abgeschlossenen String auf `stdout` aus und macht einen Zeilenvorschub.

Dateizugriff

In Standard-Pascal werden Dateitypen mit Hilfe der Konstruktion `file of <Datentyp>` deklariert. Damit ist eine sehr starre Form vorgegeben: Die Datei muß in exakt gleich strukturierten Portionen abgearbeitet werden. In der Praxis ist es aber oft sinnvoll, Daten verschiedener Typen hintereinander zu speichern. So kann etwa am Anfang einer Datei ein Vorspann stehen, der eine Typinformation enthält, und anschließend je nach Typ Daten unterschiedlicher Formate.

Dieser freien Form der Dateiverwaltung kommen in Turbo Pascal die *untypisierten Dateien* mit den Prozeduren `BlockWrite` und `BlockRead` entgegen. Wenn Sie diese Prozeduren kennen, werden Sie schnell mit den entsprechenden Bibliotheksfunktionen von C zurechtkommen.

Neben den typisierten Dateien gibt es in Pascal noch den vordefinierten Dateityp `text` für Textdateien, die in variabel lange Zeilen gegliedert sind. In C++ gibt es keinen formalen Unterschied zwischen Text- und Binärdateien. Es ist möglich, in einer Datei Text- und Binärabschnitte unterzubringen.

Wir betrachten hier die *Dateifunktionen nach der ANSI-Norm*. Alternativ dazu gibt es in Borland C++ auch die Dateifunktionen des Betriebssystems UNIX.

Wenn Sie die ANSI-Dateifunktionen verwenden, müssen Sie die Header-Datei `STDIO.H` einbinden. Neben den Funktionsdeklarationen ist dort der Datentyp `FILE` definiert. Er enthält alle Komponenten, die zur Bearbeitung einer Datei benötigt werden. Obwohl Ihnen diese Komponenten zugänglich sind (in Standard C gab es im Gegensatz zu C++ noch keinen Schutzmechanismus zum Verbergen von Typ-Details), sollten Sie den Typ `FILE` als Black Box auffassen und auf ihn nur mit den im folgenden besprochenen Funktionen zugreifen!

Öffnen einer Datei

Eine Datei wird mit der Funktion

```
FILE *fopen (const char *path, // Dateiname
const char *mode); // Art des Zugriffs
```

eröfffnet. Für die Art des Zugriffs kann eingesetzt werden:

mode	Datei wird geöffnet:
"r"	zum Lesen (read)
"w"	zum Schreiben (write)
"a"	zum Schreiben ans Ende der Datei (append)
"r+"	existierende Datei zum Lesen und Schreiben
"w+"	neue Datei zum Lesen und Schreiben
"a+"	überall lesen, schreiben nur am Ende

Ohne eine Datei zu öffnen, können folgende `FILE`-Zeiger verwendet werden:

- `stdin` Standard-Eingabe
- `stdout` Standard-Ausgabe
- `stderr` Fehlerausgabe, (mit `stdout` identisch, wenn `stdout` nicht umgeleitet wird)

Unter MS-DOS ist noch eine kleine Komplikation zu beachten. Während unter UNIX die Konvention gilt, daß der Zeilenwechsel in einer Textdatei durch das Zeichen `'\n'` (Zeilenvorschub) gekennzeichnet wird, gilt in der MS-DOS-Welt die merkwürdige Vereinbarung, daß die Textzeilen durch eine Folge von zwei Bytes, nämlich `"\r\n"` (Wagenrücklauf, Zeilenvorschub) getrennt werden. Das ist ein Relikt aus den Zeiten der Fernschreiber: Mit der Trennung von Wagenrücklauf und Zeilenvorschub war es beispielsweise möglich, eine Zeile zunächst zu schreiben und anschließend nach einem Wagenrücklauf zu unterstreichen.

Abgesehen davon, daß dieses historische Relikt unnötig Speicher verschlingt, ist sie auch ein Fallstrick beim Programmieren in C. Um die Portabilität von C-Programmen zu garantieren, die auf Textdateien zugreifen, werden Textdateien auch in den MS-DOS-Versionen von C intern im UNIX-Format behandelt. Obwohl C keinen formalen Unterschied zwischen Text- und Binärdateien macht, muß man unter MS-DOS beim Öffnen der Datei angeben, ob sie als Binärdatei oder als Textdatei behandelt werden soll. Im letzteren Fall wird Lesen aus der Datei automatisch die Byte-Folge `"\r\n"` in das Zeichen `'\n'` umgewandelt. Beim Schreiben geschieht dies in umgekehrter Richtung.

Um die Art der Umsetzung anzugeben, können Sie an den Modus-String den Buchstaben 't' für Textdateien oder 'b' für Binärdateien anhängen. Wenn Sie keinen Buchstaben anhängen, wird ein Standardwert gewählt, der in einer globalen Variablen _fmode gespeichert und auf 't' voreingestellt ist.

Lesen aus einer Datei

```
size_t fread (void *ptr, // Puffer-Adresse
size_t size, // Elementgröße in Bytes
size_t n, // Anzahl derElemente
FILE *stream);// Dateizeiger
```

Die Funktion fread (file read) entspricht der Turbo Pascal-Prozedur BlockRead. Während Sie bei BlockRead eine Puffer-Variable übergeben können, müssen Sie bei fread die Adresse explizit übergeben. Beim Lesen von Vektoren ersparen die beiden Parameter size und n eine Multiplikation.

Als Funktionswert wird die Anzahl der tatsächlich gelesenen Elemente (nicht Bytes!) zurückgegeben.

```
char *fgets (char *s, // Pufferadresse
int n, // Puffergröße
FILE *stream); // Dateizeiger
```

---

fgets (file put string) entspricht in etwa der readln-Prozedur in Pascal. Die Funktion liest bis zum nächsten Zeilenwechsel und schreibt die Zeile mit dem Zeilenende-Zeichen als String (mit '\0' abgeschlossen) in den Puffer s. Das Lesen wird vorzeitig abgebrochen, wenn die Puffergröße n nicht ausreicht.

Als Funktionswert wird die Pufferadresse s zurückgegeben. Wird beim Lesen das Dateiende überschritten, ist das Ergebnis NULL.

```
int fgetc (FILE *stream);
```

---

fgetc (file get character) liest ein Zeichen in einer Datei und gibt das gelesene Zeichen zurück, nachdem es zunächst nach unsigned char und dann nach int konvertiert wurde. Ist das Dateiende erreicht, wird die Konstante EOF zurückgegeben.

Schreiben in eine Datei

```
size_t fwrite (const void *ptr, // Puffer-Adresse
size_t size, // Elementgröße in Bytes
size_t n, // Anzahl der Elemente
FILE *stream); // Dateizeiger
```

`fwrite` entspricht `BlockWrite` in Turbo Pascal und hat die gleichen Parameter wie `fread`.

```
int fputs (const char *s, FILE *stream);
```

`fputs` ist das Gegenstück von `fgets`. Es schreibt den String `s`, der mit `'\0'` abgeschlossen sein muß, in die Datei und hängt nicht automatisch ein Zeilenwechselzeichen an.

Das folgende einfache Beispiel gibt eine Textdatei am Bildschirm aus. In der Praxis sollte man auf mögliche Fehler abfragen.

```
#include <stdio.h>

int main () {
  FILE *f;
  char s[256];
  f = fopen ("test.cpp", "rt");
  while (fgets (s, 256, f))
    fputs (s, stdout);
  fclose (f);
  return 0;
}
```

Wenn wir hier `fputs (s, stdout)` durch `puts (s)` ersetzen, wird am Bildschirm nach jeder Zeile eine Leerzeile eingefügt, weil `fgets` den Zeilenwechsel mitliest und `puts` die Ausgabe immer mit einem Zeilenvorschub beendet.

```
int fputc (int c, FILE *stream);
```

`fputc` schreibt das Zeichen `char (c)` in die Datei und gibt es als Funktionswert zurück, bei Erreichen des Dateiendes wird `EOF` zurückgegeben.

Positionieren in einer Datei

Im Gegensatz zu Standard-Pascal ist auch ein nicht-sequentieller Dateizugriff möglich. Mit `fseek` kann der Dateizeiger positioniert werden, mit `ftell` kann die Position abgefragt werden:

```
int fseek (FILE *stream, // Dateizeiger
long offset, // Relativposition
int whence); // Bezug der Relativposition
```

Für den Parameter `whence` kann eingesetzt werden:

`SEEK_SET` relativ zum Dateianfang (absolute Position)

SEEK_CUR relativ zur aktuellen Position

SEEK_END relativ zum Dateiende

fseek muß immer vor einem Wechsel zwischen Lesen und Schreiben (und umgekehrt) aufgerufen werden, außer wenn beim Lesen das Dateiende erreicht wird.

```
long ftell (FILE *stream); // gibt die absolute Position zurück
```

Schließen einer Datei

```
int fclose (FILE *stream);
```

Fehlerbehandlung

```
int feof (FILE *fp);
```

---

feof (file end of file) gibt an, ob beim letzten Zugriff das Dateiende erreicht wurde.

```
int ferror (FILE *fp);
```

---

ferror gibt an, ob beim Dateizugriff ein Fehler aufgetreten ist.

Ein Beispielprogramm zur Dateibehandlung

Das folgende Beispielprogramm zeigt einige der besprochenen Dateifunktionen und erlaubt es, mit dem nicht-sequentiellen Zugriff ein wenig zu experimentieren. Versuchen Sie zum Beispiel einmal, hinter das Dateiende zu positionieren und dann zu schreiben!

Die Funktion Auswahl dient zum Abfragen eines Menü-Buchstabens und ist auch außerhalb dieses Programms einsetzbar. Als Parameter wird ein String aus den möglichen Buchstaben übergeben. Die Funktion wartet dann solange. bis einer dieser Buchstaben eingegeben wurde und gibt ihn als Funktionswert zurück. Ein eingegebener Buchstabe wird in Großbuchstaben umgewandelt. Dann wird mit der Bibliotheksfunktion strchr geprüft, ob er im String vorkommt. Deshalb ist es wichtig, daß im String nur Großbuchstaben eingesetzt werden.

Die Funktion filesize ermittelt die Länge einer geöffneten Datei, indem sie den Dateizeiger ans Ende positioniert.

Im Hauptprogramm wird eine Datei zum Lesen und Schreiben geöffnet. Anschließend kann man menügesteuert auf beliebige Byte-Positionen positionieren und dort int-Werte lesen oder schreiben. In der Zeile

```
if (f = fopen ("test.dat", "r+b")) ...
```

wird der Wert von fopen (Dateizeiger) der Variablen f zugewiesen und gleichzeitig geprüft, ob er von NULL verschieden ist. Solche Abfragen von Zuweisungswerten sind zwar in der C-Programmierung allgemein üblich, können aber auch darauf hinweisen, daß versehentlich == mit = verwechselt wurde. Deshalb gibt der Compiler hier eine Warnung aus. Wenn Sie diesen Stil vermeiden wollen, können Sie solche Anweisungen auch in zwei Anweisungen auflösen:

```
f = fopen ("test.dat", "r+b")) ...
if (f) ...
```

Beachten Sie auch, daß die Funktion main einen Rückgabewert vom Typ int hat. Hier wird normalerweise der Wert Null zurückgegeben, bei einem Scheitern des Programms ein von Null verschiedener Wert, der unter MS-DOS mit dem Stapel-Befehl IF ERRORLEVEL ... abgefragt werden kann.

Hier beginnt das vollständige Programm:

```
// FILETEST.CPP
#include <stdio.h>
#include <conio.h>     // getch
#include <string.h>
#include <ctype.h>

char Auswahl (const char *erlaubt) {
  char z;
  do {
    z =  toupper (getch ());
    if (z == '\0')   // erweiterter IBM-Code ?
      getch ();      // nächstes Zeichen aus dem Tastaturpuffer holen
  }
  while (strchr (erlaubt, z) == NULL);
  return z;
}

long filesize (FILE *f) {
  long pos, laenge;

  pos = ftell (f);           // aktuelle Position des Dateizeigers
  fseek (f, 0L, SEEK_END);  // positioniert ans Ende der Datei
  laenge = ftell (f);
  fseek (f, pos, SEEK_SET); // setzt den Dateizeiger zurück
  return laenge;
}

int main () {
  int  n, Wert, Ergebnis;
  long Pos;
  FILE *f;
  char z;

  printf ("\n\n");
  if (f = fopen ("test.dat", "r+b")) {
```

```
    printf ("Die bestehende Datei wurde eröffnet\n");
    printf ("Dateigröße = %ld Bytes\n", filesize (f));
  }
  else if (f = fopen ("test.dat", "w+b"))
    printf ("die Datei wurde neu angelegt!\n");
  else {
    printf ("Datei kann nicht angelegt werden\n");
    return 1;
  }
  while (1) {
    Pos = ftell (f);
    printf ("aktuellePosition: %ld.   [L]esen  [S]chreiben  "
            "[P]ositionieren  [E]nde\n", Pos);
    z = Auswahl ("LSPE");
    switch (z) {
      case 'L': n = fread (&Wert, sizeof(int), 1, f);
                if (n == 1) printf ("Wert[%ld] = %d\n", Pos, Wert);
                else printf ("Lesen gescheitert (%d).\n", n);
                break;
      case 'S': printf ("Wert = ");
                scanf ("%d", &Wert); printf ("\n");
                n = fwrite (&Wert, sizeof(int), 1, f);
                if (n == 1) printf ("Wert wurde geschrieben\n");
                else printf ("Schreiben gescheitert (%d).\n", n);
                break;
      case 'P': printf ("neue Position = ");
                scanf ("%ld", &Pos); printf ("\n");
                Ergebnis = fseek (f, Pos, SEEK_SET);
                if (Ergebnis == 0)
                  printf ("Positionieren erfolgreich\n");
                else printf ("Positionieren gescheitert.\n");
                break;
      case 'E': fclose (f);
                return 0;
    }
  }
}
```

String-Funktionen

Die Funktionen `strcpy` und `strcat` haben wir bereits mehrfach verwendet. Bevor Sie eigene Funktionen zur String-Manipulation schreiben, sollten Sie einen Blick in die lange Liste der vorhandenen Bibliotheksfunktionen werfen (Datei `STRING.H`). Hier gibt es Funktionen zum Suchen, Vergleichen, zur Umwandlung von Groß- und Kleinbuchstaben und vieles mehr.

Mathematische Funktionen

Die Header-Datei `MATH.H` enthält die Deklarationen der mathematischen Funktionen nach der ANSI-Norm. Dazu gehören fast alle Pascal-Standard-Funktionen (bis auf `trunc`, `round`, `odd`, `sqr`) und viele weitere Funktionen.

Weitere wichtige ANSI-Funktionen

Hier konnte nur eine kleine Auswahl wichtiger Bibliotheksfunktionen vorgestellt werden. Es gibt eine Fülle weiterer Bibliotheksfunktionen nach der ANSI-Norm, die Ihnen viel Arbeit abnehmen können, wie etwa Quicksort (`qsort`), binäre Suche (`bsearch`),

Zufallszahlengenerator (rand), Zeit- und Datums-Routinen. Sie sollten einmal einen Streifzug durch das Referenzhandbuch machen, damit Sie später bei Bedarf das richtige Werkzeug finden.

3 Objektorientierte Programmierung

3.1 Datenabstraktion

Wenn Sie gefragt würden, warum Sie in Pascal lieber programmieren als beispielsweise in BASIC, würden Sie höchstwahrscheinlich auf das Prozedurkonzept zu sprechen kommen. Nachdem Sie sich einmal Gedanken darüber gemacht haben, wie ein bestimmtes Problem zu lösen ist, verpacken Sie die gefundene Lösung in eine Prozedur. Anschließend brauchen Sie nur noch zu wissen, *was* die Prozedur mit den übergebenen Parametern tut, nicht *wie*. Diese Möglichkeit der *Prozedurabstraktion* ermöglicht es, schrittweise immer komplexere Operationen als elementare Einheiten anzusehen und somit zur überschaubaren Behandlung immer komplexerer Probleme vorzudringen. Steht Ihnen etwa die Bibliotheksfunktion sin (für die Sinusfunktion) zur Verfügung, ist ihre Verwendung ebenso einfach, wie die der Funktion abs (für den Absolutbetrag). Mit Hilfe selbst definierter Prozeduren ist es möglich, die Programmiersprache zu erweitern und sich selbst ein immer mächtigeres Werkzeug zu schaffen.

Ein wichtiges Kriterium für einen guten Programmierstil ist die Leichtigkeit der Anpassung an veränderte Bedingungen. Wer kennt nicht jene Programme, die als holographisches Gesamtkunstwerk im Kopf ihres Schöpfers entstehen, sich später aber jedem Versuch einer Korrektur oder Anpassung mit unerklärlichem Fehlverhalten entziehen?

Die Prozedurabstraktion kommt einem in diesem Sinne guten Programmierstil entgegen. Die Implementierung einer Prozedur kann jederzeit geändert oder verbessert werden, ohne daß der Rest des Programms davon beeinträchtigt wird, solange die Anzahl und die Typen der Parameter gleich bleiben. Gerade hier stößt man aber an die Grenzen der Prozedurabstraktion. Oft stellt sich später heraus, daß bestimmte Datentypen besser anders definiert werden sollten. Dann müssen aber alle Prozeduren und Programmteile geändert werden, die auf diese Datentypen zugreifen.

Wir wollen das einfache Beispiel eines Puffers betrachten, wie er zum Beispiel als Tastaturpuffer oder als Druckerpuffer verwendet wird: Ein *Puffer* (auch *Warteschlange*, *Queue*) dient dazu, Datenelemente (in unserem Beipiel wollen wir einzelne Bytes puffern) bis zu einer maximalen Anzahl zwischenzuspeichern und bei Bedarf in richtiger Reihenfolge wieder auszugeben. Mit einem Puffer müssen also folgende Operationen durchgeführt werden können:

- den Puffer in den Anfangszustand (leer) bringen;
- angeben, ob der Puffer voll ist;
- angeben, ob der Puffer leer ist;
- ein Byte puffern;

- ein Byte ausgeben.

Prozeduren, die diese Operationen ausführen, müssen auf die Datenstruktur des Puffers zugreifen. Nun ist es aber keineswegs klar, wie man einen Datentyp `Puffer` implementiert. Man könnte eine einfach verkettete Liste verwenden mit je einem Zeiger auf den Anfang und das Ende. Ebensogut ist es aber möglich, eine zyklische Liste mit nur einem Zeiger auf das Ende zu verwenden (der Anfang ist dann einfach der Nachfolger des Endes). In vielen Fällen genügt aber auch ein Vektor fester Länge mit zwei Indexvariablen. Die Implementierung aller Puffer-Prozeduren hängt nun entscheidend von der gewählten Darstellung der Daten ab.

Ebenso, wie man mit einer Prozedur die Details einer Implementierung verbirgt, wäre es wünschenswert, auch die Details eines Datentyps zu verbergen. Wenn ein Datentyp aber seine Struktur nicht mehr preisgibt, wie kann man dann Operationen mit diesem Datentyp programmieren? Die Antwort ist: Der Datentyp muß selbst alle mit ihm möglichen Operationen ausführen können.

In unserem Beispiel eines Puffers haben wir vier Grundoperationen aufgezählt. Das Prinzip der *Datenabstraktion* besteht nun darin, die Datenstruktur Puffer zusammen mit den Grundoperationen, den sogenannten *Methoden* des Datentyps zu einem Paket so zusammenzuschnüren, daß nur noch indirekt, nämlich mit Hilfe der Methoden, auf die Datenstruktur zugegriffen werden muß. Man spricht von einem *abstrakten Datentyp*, wenn nur beschrieben wird, welche Operationen man mit den Daten ausführen kann, nicht aber wie dies geschieht. Die Methoden werden also unabhängig von einer konkreten Datenstruktur beschrieben. In der Theorie stellt man dazu ein Axiomensystem auf, das den Datentyp mit seinen Methoden eindeutig beschreibt. In der Praxis genügt es meist, die Methoden verbal zu beschreiben.

Wir könnten den abstrakten Datentyp `Puffer` etwa durch folgende konkrete Implementierung darstellen:

```
const int MAX = 10;

struct Puffer {
  char D[MAX];
  int Anfang, Ende;
};

void leere (Puffer& P) {              // Referenzparameter (wird verändert)
  P.Anfang = P.Ende = 0;
}

boolean istLeer (Puffer P) {          // Wertparameter (wird nur abgefragt)
  return P.Anfang == P.Ende;
}

boolean istVoll (Puffer P) {
  return P.Anfang == (P.Ende + 1) % MAX;
}

void lege (Puffer& P, char c) {       // Einschränkung:
  if (istVoll (P)) return;            // Fehler wird ignoriert!
  P.D[P.Ende++] = c;
  if (P.Ende >= MAX) P.Ende = 0;
}
```

```
char entnimm (Puffer& P) {
  if (istLeer (P)) return '\0';      // dto.
  char z = P.D[P.Anfang++];
  if (P.Anfang >= MAX) P.Anfang = 0;
  return z;
}
```

Hier wird eine zyklische Anordnung der Elemente des Puffers in einem Vektor der Länge `MAX` verwendet. Das erste Element hat den Index `Anfang`, das letzte den Index `Ende-1`. Ist `Anfang` gleich `Ende`, so ist der Puffer leer. Um einen vollen von einem leeren Puffer unterscheiden zu können, dürfen höchstens `MAX-1` Elemente gepuffert werden. Der Puffer ist voll, wenn dem Index `Ende` (in der zyklischen Anordnung) der Index `Anfang` direkt folgt.

Wenn man sich jetzt konsequent daran hält, nur über die Methoden auf die Objekte vom Typ `Puffer` zuzugreifen, darf die Datenstruktur später beliebig geändert werden, vorausgesetzt, die Methoden werden so angepaßt, daß sie den gleichen Axiomen genügen. Wir könnten also ohne Probleme auf eine der oben erwähnten verketteten Listendarstellungen umsteigen.

Unsere konkrete Darstellung des Typs `Puffer` entspricht übrigens genau der Implementierung des Tastaturpuffers, in den das BIOS (Basic Input/Output System) Ihres PC Tastenanschläge auf Abruf zwischenspeichert. Das BIOS bietet auch Methoden an, mit deren Hilfe man auf diesen Puffer zugreifen kann, ohne seine Struktur zu kennen.

Die oben entwickelte Implementierung des abstrakten Datentyps `Puffer` steht und fällt mit der Konsequenz, mit der wir uns an das Gebot halten, nicht direkt auf die Datenstruktur zuzugreifen. Ein wirksamer Schutz wäre es, wenn man den direkten Zugriff einfach verbieten könnte. Man spricht dann vom *Geheimnisprinzip* (*Data hiding*). Man kann zwar in jeder Programmiersprache abstrakte Datentypen realisieren; um das Geheimnisprinzip einhalten zu können, bedarf es aber einer Spracherweiterung.

3.2 Abstrakte Datentypen in C++: Das Klassenkonzept

Elementfunktionen

C++ bietet eine elegante Möglichkeit, abstrakte Datentypen unter Wahrung des Geheimnisprinzips darzustellen. Betrachten wir einfach einmal eine Darstellung unseres Puffers in C++:

```
const int MAX = 10;

struct Puffer {
public:
  boolean istLeer () const;
  boolean istVoll () const;
  void leere ();
  void lege (char c);
  char entnimm ();
private:
```

```
  char D[MAX];
  int Anfang, Ende;
};
```

Wenn Sie diese Struktur mit der zuvor definierten vergleichen, bemerken Sie folgende Neuerungen:

- Die Strukturdefinition ist in zwei Abschnitte aufgeteilt: Mit `public:` beginnt der *öffentliche Teil*, mit `private:` der *private Teil*. Elemente (Komponenten, members) des privaten Teils sind für Benutzer dieser Struktur nicht zugänglich. Läßt man diese Bereichsmarken weg, ist die ganze Struktur öffentlich.
- Die Struktur enthält neben Datenelementen auch Funktionen. Das sind die Methoden der Datenstruktur.

Die Methodenfunktionen haben jeweils einen Parameter weniger als in unserer vorherigen Definition. Das hängt mit der veränderten Sichtweise zusammen: Während man in der prozedurorientierten Programmierung einer Prozedur die Daten übergibt, sind in der objektorientierten Programmierung die Daten die zentralen Objekte. Diese Objekte können selbst alle denkbaren Operationen ausführen. Deshalb können sie ihre interne Struktur nach außen verbergen. In der Smalltalk-Sprechweise werden Botschaften an die Objekte geschickt; die Objekte veranlassen dann selbständig das Nötige. In unserem jetzigen Stadium ist das einfach nur eine andere Sichtweise des gleichen Sachverhalts. Daß daraus völlig neue Möglichkeiten erwachsen, werden Sie später sehen.

Elementfunktionen werden in der gleichen Weise angesprochen wie Datenelemente: Dem Namen des Objekts folgt, durch einen Punkt getrennt, der Name des Elements:

```
Puffer P;      // Definition eines Objekts vom Typ Puffer
P.Anfang = 0; // versuchter Zugriff auf privates Element
P.leere ();    // Elementfunktion
```

Die zweite Zeile dieses Beispiels führt zu einem Fehler, weil das Element `Anfang` privat ist. Die dritte Zeile zeigt den Aufruf der Elementfunktion `leere`. Welches Objekt geleert werden soll, sagt der Objektname P. Es gibt also bei jeder Elementfunktion einen impliziten Parameter, nämlich einen Zeiger auf das Objekt, von dem aus die Funktion aufgerufen wurde. Dieser Zeiger ist übrigens auch explizit unter dem Schlüsselwort `this` ansprechbar, was allerdings selten nötig ist.

Klassen und Strukturen

Ein abstrakter Datentyp, der so realisiert wird, wie wir es hier getan haben, wird in der objektorientierten Programmierung *Klasse* genannt. Deshalb wird in C++ statt `struct` vorzugsweise das Schlüsselwort `class` verwendet. Der einzige Unterschied zwischen `struct` und `class` besteht darin, daß in einer `struct` alle Elemente öffenlich sind, falls sie nicht als `private` erklärt werden, in einer `class` dagegen sind alle Elemente privat, wenn sie nicht als `public` erklärt werden.

Es gibt (vielleicht zu Ihrer Beruhigung) noch einen Unterschied zwischen Funktions- und

Datenelementen: Jedes Objekt einer Klasse hat seine eigenen Datenelemente. Die Funktionen werden dagegen nur einmal für die Klasse gespeichert.

Im allgemeinen sollte man sich beim Entwurf von Klassen an folgende *Faustregel* halten: Die Klassenelemente sind privat, außer wenn sie wirklich von Benutzern der Klasse verwendet werden sollen. Daraus folgt insbesondere, daß im Regelfall alle Datenelemente privat sein sollten. Der Zugriff auf die Datenelemente soll nur über spezielle Zugriffsmethoden möglich sein, die vor allem unerlaubte Zuweisungen verhindern sollen.

Betrachten wir folgendes Klassenfragment:

```
class Pixel {
public:
  void setze (int x, int y);
  int getX () const;
  int getY () const;
  void plot (int Farbe);      // zeigt den Grafik-Punkt
private:
  int px, py;
};
```

Hier sind die Koordinaten des Punkts nicht direkt zugänglich, sondern nur über die Methoden `setze`, `getX` und `getY`. Daher können wir in der Implementierung dieser Methoden dafür sorgen, daß ein `Pixel` immer innerhalb des Bildschirms liegt. Wären die Koordinaten öffentlich, müßte die Funktion `plot` jedesmal auf korrekte Koordinaten prüfen. Der Nachteil solcher *Datenkapselung* besteht darin, daß zusätzliche Zugriffsfunktionen zu schreiben sind. Diese Funktionen sind meist so einfach, daß sie als `inline` deklariert werden sollten. Dann erkauft man die zusätzliche Sicherheit nicht mit langsamerem Programmlauf.

Natürlich muß man sich nicht sklavisch an diese Regeln halten. Wenn klar ist, daß die Datenstrukturen später nicht verändert werden und es keine verbotenen Werte gibt, kann man natürlich auch auf die Privatheit der Daten verzichten. So wäre es sicher unkritisch, wenn man in einer Klasse für komplexe Zahlen den Real- und Imaginärteil einer komplexen Zahl öffentlich deklarieren würde. Borland C++ hält sich allerdings an die Regel und vereinbart die Elemente des Datentyps `complex` als `private`.

Implementierung der Elementfunktionen

In der Klassendefinition sind die Methoden zwar deklariert, aber nicht definiert. Die Implementierung einer Klasse wird außerhalb der Klassendefinition nachgeholt. Dabei wird vor jeden Funktionsnamen der Klassenname geschrieben, getrennt durch zwei Doppelpunkte. In den Funktionsdefinitionen kann direkt auf die Elemente zugegriffen werden, auch (und nur hier!) auf die privaten Elemente:

```
boolean Puffer::istLeer () const {
  return Anfang == Ende;
}

boolean Puffer::istVoll () const {
  return Anfang == (Ende + 1) % MAX;
```

```
}

void Puffer::leere () {
  Anfang = Ende = 0;
}

void Puffer::lege (char c) {
  if (istVoll ()) return;
  D[Ende++] = c;
  if (Ende >= MAX) Ende = 0;
}

char Puffer::entnimm () {
  if (istLeer ()) return '\0';
  char z = D[Anfang++];
  if (Anfang >= MAX) Anfang = 0;
  return z;
}
```

Ein vollständiges Beispiel: Bitsets

Wir wollen als zweites Beispiel ein vollständiges Programm betrachten und schrittweise die erweiterten Möglichkeiten von C++ einführen. In Pascal kann man mit Hilfe des Operators `set of` Teilmengentypen eines Ordinaltyps definieren. Betrachten Sie bitte folgendes Pascal-Programm

```
program BitsetTest;
type
  bitset = set of 0..15;

var
  i    : integer;
  a,b,c: bitset;

procedure zeigeBitset (b: bitset);
  var i: integer;
  begin
    for i := 0 to 15 do
      if i in b then write ('+')
      else write ('-');
    writeln;
  end;

begin
  a := [];         (* leere Menge *)
  b := [];
  for i := 2 to  9 do a := a + [i];
  for i := 6 to 13 do b := b + [i];
  write ('a =     '); zeigeBitset (a);
  write ('b =     '); zeigeBitset (b);
  c := a + b;  write ('a + b = '); zeigeBitset (c);
  c := a * b;  write ('a * b = '); zeigeBitset (c);
  c := a - b;  write ('a - b = '); zeigeBitset (c);
end.
```

Hier wird der (in Modula-2 übrigens schon vordefinierte) Datentyp `bitset` eingeführt. Anschließend werden die in Pascal verfügbaren Mengenoperationen Vereinigung, Durchschnitt und Mengendifferenz getestet. In C++ gibt es keine Mengentypen. Deshalb setzen wir in einer ersten Version die Mengenoperationen direkt in die entsprechenden Bit-Operationen um:

```
////////////////////////////////////////////////////////////////////
//                        BITSET0.CPP                             //
//   Konventionelle Lösung:  Direkter Zugriff auf einzelne Bits   //
////////////////////////////////////////////////////////////////////

#include <iostream.h>
#define INT_BITS 16      // abhängig vom Compiler
void zeigeBits (unsigned s) {
  cout << "{";
  for (int i=0; i<INT_BITS; i++)
    cout << (((s & (1<<i)) != 0) ? "+" : "-");
  cout << "}\n";
}

///////////////////// Test der Zugriffsmethoden: //////////////////

main () {
  unsigned a,b,c,i;
  a = b = 0;            // a und b = leere Menge
  a = ~a;               // a = Komplement (a) = {0..15}
  for (i=2; i<10; i++)
    a &= ~(1 << i);     // Elemente 2..9 entfernen: a = {0,1,10..15}
  for (i=4; i<12; i++)
    b |= 1 << i;;       // Elemente 4..11 einfügen: b = {4..11}
  cout << "\n";         // neue Zeile
  zeigeBits (a);
  zeigeBits (b);
  c = a | b;            // c = Vereinigung von a und b = {0,1,4..15}
  zeigeBits (c);
  c = a & b;            // c = Durchschnitt von a und b = {10,11}
  zeigeBits (c);
}
```

Die Nachteile eines solchen Programmierstils ohne Prozedurabstraktionen dürften klar auf der Hand liegen. Hier können sich bei jeder Mengenoperation Fehler einschleichen, weil immer wieder maschinennahe Operationen verwendet werden. Überdies sind solche Programme nur mit ausführlichen Kommentaren lesbar. Bei Kommentaren kann man sich aber nie sicher sein, ob sie wirklich den aktuellen Stand des Programms wiedergeben. Schon deshalb dürfte es vernünftiger sein, eine lesbare Version eines Programms zu verfassen. Die nächste Version zeigt, wie man in nicht objektorientierten Sprachen abstrakte Datentypen realisieren kann (indem sie das Klassenkonzept von C++ ignoriert):

```
////////////////////////////////////////////////////////////////////
//                         BITSET1.CPP                            //
//           Realisierung eines abstrakten Datentyps              //
//                  mit konventionellen Mitteln                   //
////////////////////////////////////////////////////////////////////

#include <iostream.h>
#define INT_BITS 16
#ifndef BOOLEAN  // Definition des Datentyps boolean
#define BOOLEAN
enum {false, true};
typedef int boolean;
#endif

///////////////// Definition des Datentyps bitset: ////////////////

typedef unsigned bitset;

////////////////// Zugriffsfunktionen von bitset: /////////////////

void    loesche (bitset& s)             {s = 0;}
```

```
void    incl (bitset& s, int i)         {s |= 1<<i;}
void    excl (bitset& s, int i)         {s &= ~(1<<i);}
boolean enthaelt (bitset s, int i)      {return ((s & (1<<i)) > 0);}
void    vereinige (bitset& s, bitset s1){s |= s1;}
void    schneide (bitset& s, bitset s1) {s &= s1;}
void    komplementiere (bitset& s)      {s = ~s;}

void zeige (bitset s) {
  cout << "{";
  for (int i=0; i<INT_BITS; i++) cout << (enthaelt (s, i) ? "+" : "-");
  cout << "}\n";
}

////////////////////// Test der Zugriffsmethoden: ///////////////////

main () {
  bitset a,b,c;
  loesche (a);
  loesche (b);
  for (int i=0; i<8; i++) incl (a, i);
  for (i=4; i<12; i++) incl (b, i);
  cout << "\n"; // neue Zeile
  cout << "                       a = "; zeige (a);
  cout << "                       b = "; zeige (b);
  c = a;
  vereinige (c, b);
  cout << "Vereinigung von a und b  = "; zeige (c);
  c = a;
  schneide (c, b);
  cout << "Durchschnitt von a und b = "; zeige (c);
  komplementiere (c);
  cout << "Komplement davon         = "; zeige (c);
}
```

Wir haben auf die Variablen vom Typ `bitset` hier diszipliniert zugegriffen. Das Geheimnisprinzip ist allerdings nicht verwirklicht: `bitset` ist als `unsigned` deklariert und kann beliebig manipuliert werden.

Nun sehen Sie das gleiche Programm noch einmal mit einer Klasse, in der die Daten vor direktem Zugriff verborgen sind:

```
/////////////////////////////////////////////////////////////////////
//                         BITSET2.CPP                            //
//                    Verwendung von Klassen                      //
/////////////////////////////////////////////////////////////////////

#include <iostream.h>
#define INT_BITS 16
#ifndef BOOLEAN  // Definition des Datentyps boolean
#define BOOLEAN
enum {false, true};
typedef int boolean;
#endif

/////////////////// Definition der Klasse bitset: /////////////////

class bitset {
public:
  void    loesche ();
  void    incl (int i);          // Element einfügen
  void    excl (int i);          // Element entfernen
  void    vereinige (bitset s); // Mengenvereinigung mit s bilden
  void    schneide (bitset s);  // Mengendurchschnitt mit s bilden
  void    komplementiere ();    // Mengenkomplement bilden
  boolean enthaelt (int i);     // ist i Element?
```

```
  void    zeige ();
private:
  unsigned bits;
};

/////////////// Implementierung der Klasse bitset: ///////////////

void    bitset::loesche ()           {bits = 0;}
void    bitset::incl (int i)         {bits |= 1<<i;}
void    bitset::excl (int i)         {bits &= ~(1<<i);}
boolean bitset::enthaelt (int i)     {return ((bits & (1<<i)) > 0);}
void    bitset::vereinige (bitset s) {bits |= s.bits;}
void    bitset::schneide (bitset s)  {bits &= s.bits;}
void    bitset::komplementiere ()    {bits = ~bits;}

void bitset::zeige () {
  cout << "{";
  for (int i=0; i<INT_BITS; i++) cout << (enthaelt (i) ? "+" : "-");
  cout << "}\n";
}

//////////////////// Test der Klasse bitset: ////////////////////

main () {
  bitset a,b,c;
  a.loesche ();
  b.loesche ();
  for (int i=0; i<8; i++) a.incl (i);
  for (i=4; i<12; i++) b.incl (i);
  cout << "                       a = "; a.zeige ();
  cout << "                       b = "; b.zeige ();
  c = a;
  c.vereinige (b);
  cout << "Vereinigung von a und b  = "; c.zeige ();
  c = a;
  c.schneide (b);
  cout << "Durchschnitt von a und b = "; c.zeige ();
  c.komplementiere ();
  cout << "Komplement davon         = "; c.zeige ();
}
```

Wenn nun der private Teil der Klasse `bitset` (hier nur die Variable `bits`) geändert und die Implementierung entsprechend angepaßt wird, kann die Anwendung der Klasse (hier die Funktion `main`) völlig unverändert weiterverwendet werden. Ein realistischeres Beispiel für diese Trennung zwischen abstrakter Beschreibung und konkreter Implementierung eines Datentyps finden Sie im Projektteil dieses Buchs (Abschnitt 4.6). Hier wird für einen Funktionsinterpreter eine Symboltabelle benötigt. Der hierfür beschriebene Datentyp wird alternativ als lineare Liste, und als AVL-Baum implementiert.

3.3 Konstruktoren und Destruktoren

Motivation

Selbstdefinierte Klassen bringen uns dem Ziel, mit eigenen Datentypen ebenso unkompliziert umgehen zu können, wie mit den Standard-Datentypen, einen großen Schritt näher. Bisher fehlen uns aber bei eigenen Datentypen noch ein paar Möglichkeiten.

Ein Problem ist die *dynamische Speicherplatzverwaltung*. Wenn Sie eine Variable

definieren, wird beim Eintritt des Programms in den Gültigkeitsbereich der Variablen Speicherplatz bereitgestellt und beim Verlassen des Gültigkeitsbereichs automatisch wieder freigegeben. Das war bei unserer `bitset`-Klasse kein Problem, weil der benötigte Speicher direkt als Datenelement deklariert wurde. Bei Objekten, deren Speicherbedarf nicht immer gleich ist, muß man aber zusätzlichen Speicher anfordern und darf am Ende nicht vergessen, diesen Speicher wieder freizugeben. Das folgende Turbo Pascal-Programm definiert zwei Strings und führt eine einfache Operation damit aus:

```
var
  s: string[21];
  t: string[20];
begin
  s := 'Einsturzgefahr';
  t := ' Computer ist ';
  writeln (s);
  writeln (t);
  insert (t, s, 4);
  writeln (s);
end.
```

Wir wollen jetzt versuchen, eine Klasse zu definieren, die die Turbo Pascal-Strings nachbildet. Ein String hat in Turbo Pascal eine Maximallänge, die bei der Deklaration in eckigen Klammern angegeben wird. Es wäre Speicherplatzverschwendung, wenn wir pauschal 256 Bytes reservieren würden. Stattdessen verwenden wir einen Zeiger, dem der wirklich benötigte Speicher zur Verfügung gestellt wird.

Konstruktoren

C++ erlaubt es, zu jeder Klasse einen oder mehrere Konstruktoren zu definieren. Ein *Konstruktor* ist eine Methode, die automatisch aufgerufen wird, wenn das Programm in den Gültigkeitsbereich eines Objekts eintritt. Um verschiedenartige Objekte einer Klasse definieren zu können, darf der Konstruktor auch eine nichtleere Parameterliste haben. In unserem Beispiel könnten wir einen Parameter verwenden, in dem die Maximallänge des Strings angegeben wird.

Wie wird nun ein Konstruktor definiert? Formal ist er eine Elementfunktion ohne Funktionswert, deren Name der Klassenname ist. Allerdings wird bei einem Konstruktor auch der Pseudo-Typ `void` weggelassen, der sonst für einen nicht vorhandenen Funktionswert angegeben wird. Sie sehen hier eine (unvollständige) String-Klasse mit einem Konstruktor:

```
class string {
public:
  string (int n = 255);   // Konstruktor
  ...
private:
  char *s;
  int aktLg;
  int maxLg;
};
```

Hier hat der Konstruktor ein Argument mit einem Standardwert. Genau genommen, haben

wir damit zwei Konstruktoren definiert, nämlich einen ohne Argument und einen mit einem `int`-Argument. Damit können wir Strings fast wie in Pascal definieren. Die Definition

```
string s1, s2 (20);
```

soll einen String `s1` der Maximallänge 255 (Standard) und einen String `s2` der Maximallänge 20 erzeugen. Die Implementierung könnte so aussehen:

```
string::string (int n) {
  if (n < 1) n = 1;              // Maximallänge 0 sinnlos
  else if (n > 255) n = 255;
  aktLg = 0;
  maxLg = n;
  s = new char[n+1];
}
```

Unsere Implementierung unterscheidet sich von der in Turbo Pascal darin, daß die Länge in einem `int`-Element gespeichert wird. Deshalb sind wir hier nicht an die Beschränkung der Maximallänge auf 255 gebunden.

Wenn wir bei der Implementierung die Bibliotheksfunktionen zur String-Verarbeitung verwenden wollen, müssen wir dem String noch ein Nullbyte anhängen können. Deshalb werden in der letzten Zeile `n+1` Bytes reserviert.

Destruktoren

Mit dem Bereitstellen des Speicherplatzes allein ist es allerdings nicht getan. Wir müssen auch dafür sorgen, daß beim Verlassen des Gültigkeitsbereichs des Objekts der dynamische Speicher wieder freigegeben wird. Hierfür dient ein Destruktor. Ein *Destruktor* wird wie ein Konstruktor geschrieben, nur wird das Tilde-Zeichen (~) davorgestellt. Destruktoren dürfen keine Parameter haben. Da unser Konstruktor der `string`-Klasse zusätzlichen Speicherplatz anfordert, benötigen wir einen Destruktor, der den Speicher wieder freigibt. Es folgt ein komplettes Programm, in dem exemplarisch die String-Prozedur `insert` wie in Turbo Pascal implementiert ist:

```
////////////////////////////////////////////////////////////////////
//                                                                //
//           TPSTRING.CPP - Turbo Pascal-Strings in C++           //
// Beispiel für die Verwendung von Konstruktoren und Destruktoren //
//                                                                //
////////////////////////////////////////////////////////////////////
#include <iostream.h>
#include <string.h>

class string {
public:
  string (int n = 255);                          // Konstruktor
  ~string ();                                    // Destruktor
  int length ();
  void insert (string& Einfg, int Index);
  void setze (char *s);
  void writeln ();
private:
  char *s;
  int aktLg;
```

```
  int maxLg;
};

string::string (int n) {
  if (n < 1) n = 1;
  else if (n > 255) n = 255;
  aktLg = 0;
  maxLg = n;
  s = new char[n+1];
}

string::~string () {
  delete s;
}

int string::length () {
  return aktLg;
}

void string::insert (string& Einfg, int Index) {
  int AnfLg;           // Länge des Anfangsstücks (bleibt stehen)
  int EndLg;           // Länge des Endstücks (wird verschoben)
  int EinfgLg;         // Länge des einzufügenden Strings
  AnfLg = Index - 1;   // Indizierung in Pascal ab 1
  if (AnfLg < 0) AnfLg = 0;                  // Fehler abfangen
  else if (AnfLg > aktLg) AnfLg = aktLg;     // dto.
  EinfgLg = Einfg.length ();
  // falls Einfg keinen Platz hat: beschneiden:
  if (AnfLg + EinfgLg > maxLg) EinfgLg = maxLg - AnfLg;
  EndLg = aktLg - AnfLg;
  // falls verschobenes Endstück keinen Platz hat: beschneiden:
  if (AnfLg + EinfgLg + EndLg > maxLg) EndLg = maxLg - AnfLg - EinfgLg;
  memmove (s+AnfLg+EinfgLg, s+AnfLg, EndLg);       // Ende aufrücken
  memmove (s+AnfLg, Einfg.s, EinfgLg);    // Einfg in Lücke kopieren
  aktLg = AnfLg + EinfgLg + EndLg;
}

void string::setze (char *source) {
  int n = strlen (source);
  if (n > maxLg) n = maxLg;            // notfalls beschneiden
  memmove (s, source, n);
  aktLg = n;
}

void string::writeln () {
  s[aktLg] = '\0';
  cout << s << '\n';
}

main () {
  string s (21), t (20);
  s.setze ("Einsturzgefahr");
  t.setze (" Computer ist ");
  cout << "\n";
  s.writeln ();
  t.writeln ();
  s.insert (t, 4);
  s.writeln ();
}
```

Standard-Konstruktoren und -Destruktoren

Wenn Sie eine Klasse ohne Konstruktor definieren, erzeugt C++ automatisch einen Konstruktor ohne Argumente, den *Standard-Konstruktor*. Dieser Standard-Konstruktor tut einfach das, was Sie von Pascal her gewohnt sind: Den Datenelementen wird beim

Eintritt des Programms in den Gültigkeitsbereich des Objekts der entsprechende Speicherplatz bereitgestellt. Dabei werden die Daten nicht initialisiert, sondern haben unvorhersagbare Anfangswerte. Wenn Sie wollen, daß Objekte automatisch einen bestimmten Anfangswert bekommen, können Sie dafür einen eigenen Konstruktor definieren, der die Elemente nach Wunsch initialisiert.

Wenn Sie keinen Destruktor definieren, wird der Standard-Destruktor verwendet, der einfach nur die direkten Datenelemente beseitigt. Sobald Sie aber irgendeinen Konstruktor definieren, ist der Standard-Konstruktor nicht mehr verfügbar.

Kopier-Konstruktoren

Noch ein zweiter Typ von Konstruktoren wird vom Compiler automatisch erzeugt, wenn Sie ihn nicht explizit definieren, nämlich der *Kopier-Konstruktor*. Der Kopier-Konstruktor dient der Initialisierung eines Objekts mit dem Wert eines anderen Objekts der gleichen Klasse. Wenn Definitionen, wie

```
int a = 7;
int b = a;
```

möglich sind, warum sollte man dann nicht auch

```
string s (20);
string t = s;
```

schreiben dürfen? Sie dürfen es, nur ist die Wirkung nicht so, wie sie sein sollte. Da wir in der `string`-Klasse keinen Kopier-Konstruktor definiert haben, wird hier der *Standard-Kopier-Konstruktor* verwendet. Dieser erzeugt eine Kopie aller Datenelemente. Das bedeutet aber, daß auch der Zeiger `s.s` nach `t.s` kopiert wird. Damit teilen sich die Strings `s` und `t` den gleichen Speicherplatz. Das hat recht unangenehme Folgen: Wenn ein String geändert wird, wird der Inhalt des anderen mitgeändert, die Länge (`aktLg`) aber nicht. Außerdem wird später beim Aufruf der Destruktoren versucht, den gleichen Speicherplatz zweimal freizugeben.

Hier müssen wir also einen Kopier-Konstruktor definieren. Ist `X` eine Klasse, so hat der Kopier-Konstruktor die Form `X (const X&)`. Wir ergänzen also in der Klassendefinition die Zeile

```
  string (const string& t);
```

Wie sieht nun die Implementierung aus? Hätten wir keinen Kopier-Konstruktor definiert, würde automatisch einer erzeugt, der alle Datenelemente kopiert:

```
string::string (const string& t) {
  maxLg = t.maxLg;
  aktLg = t.aktLg;
  s = t.s;
}
```

Wir wollen aber, daß die Kopie eigenen Speicherplatz für einen String bekommt und schreiben daher:

```
string::string (const string& t) {
  maxLg = t.maxLg;
  aktLg = t.aktLg;
  s = new char[maxLg+1];
  memmove (s, t.s, aktLg);
}
```

Vektoren von Objekten

Wir haben gesehen, daß bei der Definition eines Objekts dem Konstruktor Parameter übergeben werden können. In der Definitions-Anweisung

```
string s (20);
```

wird dem Konstruktor für die Klasse `string` der aktuelle Parameter 20 übergeben. Solche Parameterübergaben sind in C++ nur in einfachen Definitionen möglich, nicht aber in Vektordefinitionen. Vektoren von Objekten können deshalb nur definiert werden, wenn für die Klasse entweder gar kein Konstruktor oder ein Konstruktor ohne Parameter definiert wurde. Im ersten Fall wird für jedes Element der Standard-Konstruktor verwendet.

Die Definitions-Anweisung

```
string s[10];
```

erzeugt einen Vektor aus 10 Elementen vom Typ `string`. Dabei wird 10 mal der Konstruktor ohne Argumente aufgerufen. In unserer Implementierung wird also für jeden String die maximale Länge von 255 Bytes reserviert.

Wenn wir einen Vektor aus 10 Strings mit der Maximallänge 20 benötigen, ist das mit unserer `string`-Klasse nicht möglich. In solchen Fällen muß man die Klasse anders konzipieren. Wir könnten etwa die Reservierung des Speicherplatzes für den String aus dem Konstruktor entfernen und hierfür eine separate Elementfunktion einführen:

```
class string {   // Alternative für Vektoren von Strings
public:
  initialisiere (int n = 255);
  ~string ();
  ...
private:
  char *s;
  int aktLg;
  int maxLg;
};

int main () {
  string S[10];
  for (int i=0; i<10; i++) S[i].initialisiere (20);
  ...
}
```

Damit gehen allerdings die Vorteile des Konstruktors verloren. Jetzt besteht die Gefahr, daß die Initialisieruung versehentlich mehrfach aufgerufen wird. Der Destruktor der Klasse `string` kann unverändert weiterverwendet werden.

3.4 Friend-Funktionen

Manchmal ist es sinnvoll, Funktionen mit Argumenten einer Klasse außerhalb der Klasse zu definieren. Unsere Bitset-Klasse bietet zum Beispiel eine Methode

```
void bitset::schneide (bitset s);
```

Hier wird einer Menge der Durchschnitt dieser Menge mit einer anderen Menge zugewiesen. Manchmal wäre aber auch eine Funktion sinnvoll, die den Durchschnitt zweier Mengen als Ergebnis zurückgibt. Man könnte auch hier eine Methode

```
bitset bitset::Durchschnitt (bitset s);
```

innerhalb der Klasse definieren. Wollen wir den Durchschnitt d zweier Bitsets a und b berechnen, könnten wir wahlweise

```
d = a.Durchschnitt (b);
```

oder

```
d = b.Durchschnitt (a);
```

schreiben. Diese Asymmetrie ist hier ästhetisch unbefriedigend, denn beide Argumente sind ja völlig gleichberechtigt. Deshalb bietet sich hier eine Funktion außerhalb der Klasse an, die zwei Klassenargumente annimmt.

```
bitset Durchschnitt (bitset a, bitset b) {  // Durchschnitt
  bitset s;
  s.bits = a.bits & b.bits;
  return s;
}
```

Leider ist das aber so nicht möglich, weil das Element `bits` privat ist. Würde man aber deshalb `bits` als `public` erklären, hieße das, auf alle Vorteile des Geheimnisprinzips zu verzichten.

C++ bietet hier die Möglichkeit, gezielte Ausnahmen der Privatheit zu erklären: Eine Klasse kann einer Funktion außerhalb der Klasse den Zugriff auf ihre privaten Elemente erlauben. Hierzu wird innerhalb der Klassendefinition eine Deklaration der zugriffsberechtigten Funktion eingefügt und das Schlüsselwort `friend` davorgestellt. In unserem Beispiel sähe das so aus:

```
class bitset {
  ...
```

```
  friend bitset Durchschnitt (bitset a, bitset b);
  ...
};
```

Es ist auch erlaubt, eine andere Klasse als `friend` zu erklären:

```
class A {
  ..
  friend class B;
  ...
};
```

Mit dieser Erklärung haben alle Funktionen der Klasse B das Zugriffsrecht auf die privaten Elemente der Klasse A.

Es gibt keine Vorschrift, an welcher Stelle in einer Klassendefinition die `friend`-Erklärungen stehen sollen. Da dies Informationen sind, die für den Benutzer der Klasse unwesentlich sind, empfiehlt es sich, sie im privaten Teil der Klassendefinition unterzubringen.

Zwei Klassen können sich auch gegenseitig das Zugriffsrecht geben. Hier muß die Klasse, die im Text zuerst erscheint, die andere Klasse als Freund erklären. Da die andere Klasse hier aber noch nicht definiert ist, wird eine vorherige *Deklaration* der zweiten Klasse benötigt, wie das folgende konstruierte Beispiel zeigt:

```
class LONG;       // Deklaration der Klasse LONG

class INT {
public:
  void addiere (LONG l);
private:
  friend class LONG;
  int n;
};

class LONG {      // Definition der Klasse LONG
public:
  void addiere (INT i);
private:
  friend class INT;
  long n;
};

void INT::addiere (LONG l) {n += int (l.n);}
void LONG::addiere (INT i) {n += long (i.n);}
```

3.5 Überladen von Operatoren

Sie wissen bereits, daß in C++ Funktionen überladen werden dürfen. Da Operatoren nur eine andere Schreibweise für Funktionen sind, ist es naheliegend, auch das Überladen von Operatoren zu erlauben. Der Compiler kommt ja ohnehin schon mit den vordefinierten arithmetischen Operatoren zurecht, die je nach Datentyp völlig verschiedene Funktionen aufrufen.

Deklaration überladener Operatoren

C++ erlaubt das *Überladen von Operatoren*, allerdings mit zwei Einschränkungen:

- Es dürfen nur die in der Sprache schon vorhandenen Operatoren (bis auf wenige Ausnahmen) überladen werden.
- Mindestens ein Argument des Operators muß ein Klassentyp sein.

Bei der Deklaration überladener Operatoren wird der Operator formal wie eine gewöhnliche Funktion dargestellt, indem ihm das Schlüsselwort `operator` vorangestellt wird. Wenn Sie etwa eine Klasse `Zahl` (z. B. für große Zahlen) definieren, können Sie die Summenfunktion für diesen Typ so deklarieren:

```
Zahl operator + (Zahl z1, Zahl z2);
```

Verwendet wird dieser überladene Operator dann wie üblich.

Es gibt zwei verschiedene Möglichkeiten, überladene Operatoren zu definieren:

- *Definition außerhalb der Klasse*
  Falls der Operator auf ein privates Element der Klasse zugreift (das ist der Normalfall!), muß die Klasse ihn als `friend` deklarieren.
- *Definition innerhalb der Klasse*
  Überladene Operatoren können auch als Elementfunktion definiert werden, falls das erste Argumet eine Referenz auf ein Objekt der Klasse ist. In diesem Fall wird das erste Argument nicht explizit angegeben.

Im folgenden (sinnlosen) Beispiel sehen Sie beide Arten der Definition von überladenen Operatoren:

```
class Zahl {
public:
  void operator += (Zahl b);
private:
  friend void operator -= (Zahl& a, Zahl b);
  friend Zahl operator + (Zahl a, Zahl b);
  int n;
};
```

Der Operator += ist innerhalb der Klasse definiert, -= außerhalb der Klasse. Hier sollen die verschiedenen Möglichkeiten demonstriert werden. In der Praxis sollte man natürlich einen einheitlichen Stil wählen.

Für den Operator + haben wir keine andere Wahl, als ihn außerhalb der Klasse zu definieren, weil sein erstes Argument keine Referenz auf die Klasse ist.

Es folgt die Implementierung der Klasse:

```
void Zahl::operator += (Zahl b)    {n += b.n;}
void operator -= (Zahl& a, Zahl b) {a.n -= b.n;}
```

```
Zahl operator + (Zahl a, Zahl b) {
  Zahl s;
  s.n = a.n + b.n;
  return s;
}
```

Anwendung auf die Klasse bitset

Wir setzen die neu gewonnenen Möglichkeiten nun für eine abschließende Version der Klasse `bitset` ein. Dies sind die Neuerungen:

- Ein Konstruktor (ohne Argumente) sorgt dafür, daß jedes Objekt mit der leeren Menge initialisiert wird.
- Es werden implizite Typkonvertierungen von `int` nach `bitset` und umgekehrt eingeführt.
- Die von Pascal gewohnten Operatoren für Vereinigung, Durchschnitt und Differenzmenge werden übernommen und durch weitere naheliegende Operatoren für das Einfügen und Löschen von Elementen, Komplement, und Elementrelation ergänzt. Den Pascal-Operator `in` können wir nicht übernehmen, weil in C++ leider keine neuen Operatorsymbole eingeführt werden können. Stattdessen überladen wir hier das Kleiner-Zeichen (<).

In der Implementierung werden die Mengenoperationen durch die entsprechenden Bit-Operatoren ersetzt.

Als praktische Anwendung dient die Funktion `TastenDemo`, in der der Zustand der Umschalttasten, der Strg- (Ctrl-) und der Alt-Taste auf einem IBM-kompatiblen PC abgefragt wird. Eigentlich sollte hierzu die entsprechende BIOS-Routine aufgerufen werden (in Borland C++ auch unter `bioskey (2)` verfügbar), um Portabilität zu garantieren. Hier soll aber ein konkretes Beispiel für Referenzen und für `volatile`-Variablen gegeben werden. Deshalb wird die Speicherstelle 417H abgefragt, die ständig in ihren einzelnen Bits die zur Zeit gedrückten Umschalttasten wiedergibt. `FlagPtr` ist ein Zeiger auf diese Speicherstelle. Die Adresse wird mit der Funktion `MK_FP` ("MaKe Far Pointer") aus Segment und Offset gebildet:

```
bitset far *FlagPtr = (bitset far *) MK_FP (0x0040, 0x17);
```

Das Attribut `far` bewirkt, daß auch bei einem kleinen Datenmodell hier eine vollständige Adressierung (mit Segment und Offset) vorgenommen wird. Um die Dereferenzierung dieses Zeigers zu verbergen, definieren wir eine `bitset`-Referenz auf diese Adresse:

```
volatile bitset far &Flags = *FlagPtr;
```

Die Variable `Flags` ändert sich bei Tastendrücken außerhalb der Kontrolle unseres Programms (durch eine Interrupt-Routine). Dies könnte dem Programm entgehen, wenn der Compiler das Programm optimiert und den Wert in ein Prozessor-Register kopiert. Deshalb muß mit dem Attribut `volatile` verhindert werden, daß der Compiler erneutes Lesen aus

dem Hauptspeicher wegoptimiert.

```
///////////////////////////////////////////////////////////////////////
//                          BITSET3.CPP                              //
//         Mit Konstruktoren, und Überladen von Operatoren           //
///////////////////////////////////////////////////////////////////////

#include <iostream.h>
#include <dos.h>        // nur für Tastendemo ()
#define INT_BITS 16
#ifndef BOOLEAN
#define BOOLEAN
enum {false, true};
typedef int boolean;
#endif

class bitset {
public:
  bitset ();                // Konstruktor (Initialisierung)
  void operator += (int i); // Element einfügen
  void operator -= (int i); // Element entfernen
  boolean leer ();          // Leere Menge
private:
  friend boolean operator < (int i, bitset s);      // Element
  friend bitset operator + (bitset s1, bitset s2);  // Vereinigung
  friend bitset operator * (bitset s1, bitset s2);  // Durchschnitt
  friend bitset operator - (bitset s1, bitset s2);  // Mengendiff.
  friend boolean operator < (bitset s1, bitset s2); // Teilmenge
  friend bitset operator ~ (bitset s1);             // Komplement
  friend ostream& operator << (ostream& s, bitset b);
  unsigned bits;
};

inline bitset::bitset ()                {bits = 0;}
inline void bitset::operator += (int i) {bits |= 1<<i;}
inline void bitset::operator -= (int i) {bits &= ~(1<<i);}
inline boolean bitset::leer ()          {return bits == 0;}

boolean operator < (int i, bitset s) {      // Element
  return ((s.bits & (1<<i)) > 0);
}

bitset operator + (bitset s1, bitset s2) {  // Vereinigung
  bitset s;
  s.bits = s1.bits | s2.bits;
  return s;
}

bitset operator * (bitset s1, bitset s2) {  // Durchschnitt
  bitset s;
  s.bits = s1.bits & s2.bits;
  return s;
}

bitset operator - (bitset s1, bitset s2) {  // Mengendifferenz
  bitset s;
  s.bits = s1.bits & ~s2.bits;
  return s;
}

boolean operator < (bitset s1, bitset s2) { // Teilmenge
  return (s1.bits & ~s2.bits) == 0; // s1-s2 = Ý
}

bitset operator ~ (bitset s1) {             // Komplement
  bitset s;
  s.bits = ~s1.bits;
  return s;
```

```
}

ostream& operator << (ostream& s, bitset b) {
  s << "{";
  for (int i=0; i<INT_BITS; i++) s << ((i < b) ? "|" : "·");
  s << "}";
  return s;
}

void BitsetDemo () {
  bitset a,b,c;
  for (int i=2; i<10; i++) a += i;
  for (i=6; i<14; i++) b += i;
  cout << "    a = " << a << "\n";
  cout << "    b = " << b << "\n";
  c = a + b;  cout << "a + b = " << c << "\n";
  c = a * b;  cout << "a * b = " << c << "\n";
  c = a - b;  cout << "a - b = " << c << "\n";
  c = ~ a;    cout << "   ~a = " << c << "\n";
}

void TastenDemo () {
  enum shiftstate {
    RShift, LShift, Ctrl, Alt, ScrollLock, NumLock, CapsLock, Insert
  };
  bitset far *FlagPtr = (bitset far *) MK_FP (0x0040, 0x17);
  volatile bitset far &Flags = *FlagPtr;
  bitset alteFlags, neueFlags, AbbruchFlags, gesuchteFlags;

  cout << "\n Anzeige der Tasten Shift, Ctrl, Alt "
          "(Ende: beide Shift-Tasten)\n";
  gesuchteFlags += RShift;
  gesuchteFlags += LShift;
  gesuchteFlags += Ctrl;
  gesuchteFlags += Alt;
  AbbruchFlags += RShift;
  AbbruchFlags += LShift;
  do {
    neueFlags = Flags;
    neueFlags = neueFlags * gesuchteFlags;
    if (neueFlags != alteFlags) {
      alteFlags = neueFlags;
      cout << "\ngedrückte Umschalttasten: ";
      if (neueFlags.leer ())  cout << "-----";
      if (RShift < neueFlags) cout << "RShift ";
      if (LShift < neueFlags) cout << "LShift ";
      if (Ctrl < neueFlags)   cout << "Ctrl ";
      if (Alt < neueFlags)    cout << "Alt ";
    }
  }
  while (! (AbbruchFlags < neueFlags));
}

main () {
  BitsetDemo ();
  TastenDemo ();
}
```

Hinweise für das Überladen der einzelnen Operatoren

- *Allgemeines*
  Beim Überladen von Operatoren haben Sie die Freiheit, jedem Operator im Zusammenhang mit einer Klasse eine beliebige Bedeutung zu geben. Dabei sollten aber in jedem Fall Erwartungen berücksichtigt werden, die sich aus der üblichen

Bedeutung des Operators aufdrängen. So kann es sinnvoll sein, den Operator + wie in Pascal für die Verkettung von Strings oder für die Vereinigung von Mengen zu verwenden. Wenn keine naheliegende Analogie zur üblichen Verwendung eines Operators besteht, sollte man lieber auf Operatoren verzichten und eine sinnvoll benannte Funktion verwenden.

- *Der Zuweisungoperator (=)*
  Wenn der Zuweisungsoperator nicht überladen wird, wird ein Standard-Mechanismus verwendet: Alle Datenelemente werden einzeln zugewiesen. Das ist oftmals nicht das gewünschte, nämlich meist dann, wenn die Klasse Zeigerelemente enthält. Wenn statt der Zeiger die Daten kopiert werden sollen, auf die die Zeiger zeigen, muß der Zuweisungsoperator überladen werden.
- *Die arithmetischen Zuweisungsoperatoren (+=, -= usw.)*
  Diese Operatoren ergeben sich nicht automatisch aus den entsprechenden arithmetischen Operatoren und dem Zuweisungsoperator: Wenn Sie = und + überladen haben, müssen Sie den Operator += trotzdem explizit überladen, wenn Sie ihn verwenden wollen. Trotz dieser Unabhängigkeit sollten Sie diese Operatoren so implementieren, daß sie die Kombination des arithmetischen Operators und der Zuweisung darstellen.
- *Die Inkrementierungsoperatoren (++, --)*
  Diese Operatoren können nach dem Überladen sowohl als Präfix- wie auch als Postfix-Operatoren verwendet werden. Die Wirkung ist aber in beiden Fällen die gleiche.
- *Die Gleichheitsrelation (==) und die Ungleichheitsrelation (!=)*
  Im Gegensatz zum Zuweisungsoperator gibt es keine Standard-Behandlung der Gleichheits- bzw. Ungleichheitsrelation. Wenn Sie Klassenobjekte auf Gleichheit oder Ungleichheit prüfen möchten, müssen Sie diese Operatoren überladen, es sei denn, Sie definieren einen Konvertierungsoperator in einen Standard-Datentyp, und der Vergleich der konvertierten Werte liefert das von Ihnen gewünschte Ergebnis.
- Der Ungleichheitsoperator ist auch nicht automatisch das logische Gegenteil des Gleichheitsoperators. Es müssen also, wenn sie benötigt werden, stets beide Operatoren überladen werden.
- *Der Vektorklammer-Operator (`[]`)*
  Dieser Operator wird verwendet, um bei selbstdefinierten Vektor- oder Matrixtypen die übliche Schreibweise anwenden zu können. Der Operator `[]` ist ein zweistelliger Operator, der aber in seiner Syntax etwas aus dem Rahmen fällt. Während die üblichen zweistelligen Operatoren zwischen den beiden Operanden stehen, verteilt sich der `[]`-Operator: das Zeichen `[` steht zwischen den Operanden, das Zeichen `]` steht hinter dem zweiten Operanden.

  Wenn Sie den Operator sowohl in Ausdrücken wie `a = b[i]`, als auch in Ausdrücken wie `b[i] = a` anwenden wollen, muß die Operatorfunktion eine Referenz zurückgeben (`b[i]` muß ein L-Wert sein)! Im folgenden Abschnitt finden Sie hierzu ein Beispiel, in dem große Vektoren auf eine temporäre Datei ausgelagert werden.

Der Unterschied zwischen Zuweisung und Initialisierung

Es ist wichtig, daß Sie sich den Unterschied zwischen Zuweisungen und Initialisierungen klarmachen: Bei der *Zuweisung* bekommt ein schon initialisiertes Objekt einen neuen Wert, bei der *Initialisierung* wird ein neues Objekt mit Hilfe des Kopier-Konstruktors erst erzeugt. Bei den Standard-Datentypen ist der Unterschied unerheblich:

```
int i = 1; // i wird mit 1 initialisiert
int j;     // j wird nicht initialisiert (unbestimmter Wert)
j = 2;     // jetzt bekommt j den Wert 2;
```

Ganz anders sieht es aber oft bei Klassentypen aus. Betrachten wir nochmals die `string`-Klasse. Das Problem der Initialisierung durch einen anderen String haben wir mit Hilfe des Kopier-Konstruktors gelöst. Für den neuen String wird Speicherplatz bereitgestellt und der Inhalt des anderen Strings kopiert. Bei einer Zuweisung eines Strings an einen anderen haben beide schon ihren eigenen Speicher. Jetzt sollte nur der Inhalt kopiert werden. In der bisherigen Form der String-Klasse funktioniert das aber nicht, weil der Standard-Mechanismus für die Zuweisung verwendet wird: Es wird einfach das Zeiger-Element `s` kopiert mit den oben diskutierten Folgen. Deshalb muß hier auch die Zuweisung überladen werden:

```
class string {
public:
  string& operator = (string& source);
  ...
};

string& string::operator = (string& source) {
  int n = source.aktLg;
  if (n > maxLg) n = maxLg;
  memmove (s, source.s, n);
  aktLg = n;
  return *this;
}
```

Hier liefert die Zuweisung eine Referenz auf das Zugewiesene zurück, wie dies in C++ üblich ist. Nur so können auch überladene Zuweisungen verkettet werden.

Zum Vergleich folgt nochmals die Implementierung des Kopier-Konstruktors:

```
string::string (const string& t) {
  maxLg = t.maxLg;
  aktLg = t.aktLg;
  s = new char[maxLg+1];
  memmove (s, t.s, aktLg);
}
```

Die Zuweisung ähnelt zwar sehr dem Kopier-Konstruktor. Beide kopieren nicht das Zeiger-Element sondern die Daten, auf die es zeigt. Während aber der Kopier-Konstruktor erst ein Objekt erzeugen muß, existiert das Objekt bei der Zuweisung schon.

Arithmetik mit rationalen Zahlen

Bei der Division zweier ganzer Zahlen erhält man *rationale Zahlen*. Näherungsweise läßt sich eine rationale Zahl durch eine Gleitkommazahl darstellen. Die Darstellung wird aber ungenau, wenn die Gleitkomma-Darstellung periodisch ist. Es folgen einige Beispiele:

```
float x = 1/3; // Rundungsfehler, weil 1/3 periodischer Dualbruch
float y = 1/4; // kein Fehler
float z = 1/5; // Rundungsfehler, weil 1/5 periodischer Dualbruch
```

Bei der dritten Zuweisung ergibt sich im Dezimalsystem ein abbrechender Bruch (0,2). Im Computer wird die Zahl jedoch im Dualsystem dargestellt und ergibt den periodischen Dualbruch 0,00110011...

In manchen Anwendungen (zum Beispiel in der Computergrafik bei der Speicherung von Schnittpunkten zweier Geraden) wird eine exakte Darstellung von rationalen Zahlen benötigt. Für solche Anwendungen wollen wir eine Klasse `rational` entwickeln, die rationale Zahlen exakt darstellt, indem sie Zähler und Nenner speichert. Das erschwert natürlich die arithmetischen Operationen etwas.

Zähler und Nenner werden in den privaten Elementen `p` und `q` vom Typ `long` gespeichert. Zunächst sehen Sie die Header-Datei:

```
//////////////////////////// RATIONAL.H /////////////////////////////
//                                                                  //
//              Arithmetik mit rationalen Zahlen                    //
//                                                                  //
//////////////////////////////////////////////////////////////////////
#ifndef RATIONAL_H
#define RATIONAL_H
#include <iostream.h>

class rational {
public:
  rational (long Zaehler = 0, long Nenner = 1);
private:
  friend rational operator - (rational r);           // unäres Minus
  friend rational operator + (rational r1, rational r2);
  friend rational operator - (rational r1 , rational r2);
  friend rational operator * (rational r1, rational r2);
  friend rational operator / (rational r1, rational r2);
  friend rational& operator += (rational& r, rational r2);
  friend rational& operator -= (rational& r, rational r2);
  friend rational& operator *= (rational& r, rational r2);
  friend rational& operator /= (rational& r, rational r2);
  friend int operator == (rational& r1, rational& r2);
  friend int operator != (rational& r1, rational& r2);
  friend ostream& operator << (ostream& s, rational r);
  long p, q;
};

#endif RATIONAL_H
```

Nun folgt die Implementierung der Klasse `rational`. Hier wird ein möglicher Überlauf bei der Arithmetik mit `long`-Zahlen ignoriert. In Borland C++ gibt es keine Möglichkeit, Bereichsüberschreitungen bei Ganzzahltypen abzufangen. Man könnte beispielsweise die

`long`-Multiplikation durch folgende Funktion ersetzen:

```
long Produkt (long a, long b) {
  if (LONG_MAX / a < b) {
    cerr << "\nÜberlauf bei Rational-Arithmetik.\n";
    exit (1);
  }
  else return a * b;
}
```

Das ginge allerdings sehr auf Kosten der Effizienz. Für den praktischen Einsatz könnte man auch eine Assembler-Routine für die Multiplikation schreiben, die das Überlauf-Flag des Prozessors abfragt.

Der Konstruktor bringt die rationale Zahl auf eine eindeutige *Normalform*: Zähler und Nenner sind gekürzt, und der Nenner ist positiv. Auch alle Operatoren bringen ihre Ergebnisse immer in diese Normalform. Das ist die Voraussetzung für die einfache Implementierung des Operators ==. Zum Kürzen wird der bekannte *Euklidische Algorithmus* verwendet.

```
////////////////////////// RATIONAL.CPP ////////////////////////////
//                                                                //
//              Arithmetik mit rationalen Zahlen                  //
//                                                                //
////////////////////////////////////////////////////////////////////

#include "rational.h"
#include <stdlib.h>

long ggT (long a, long b) {           // Euklidischer Algorithmus,
  while (b > 0) {                     // Voraussetzung: a, b >= 0
    long m = a % b;
    a = b;
    b = m;
  }
  return a;
}

rational::rational (long Zaehler, long Nenner) {
  p = Zaehler;
  q = Nenner;
  if (q < 0) {p = -p; q = -q;}  // Nenner immer positiv
  long t = ggT (labs (p), q);
  if (t > 1) {p /= t; q /= t;}  // kürzen, falls möglich
}

rational operator - (rational r) {
  r.p = -r.p;
  return r;
}

rational operator + (rational r1, rational r2) {
  return rational (r1.p * r2.q + r2.p * r1.q, r1.q * r2.q);
}

rational operator - (rational r1, rational r2) {
  return r1 + (-r2);
}

rational operator * (rational r1, rational r2) {
  return rational (r1.p * r2.p, r1.q * r2.q);
}
```

```
rational operator / (rational r1, rational r2) {
  return rational (r1.p * r2.q, r1.q * r2.p);
}

rational& operator += (rational& r1, rational r2) {
  r1 = r1 + r2;
  return r1;
}

rational& operator -= (rational& r1, rational r2) {
  r1 = r1 - r2;
  return r1;
}

rational& operator *= (rational& r1, rational r2) {
  r1 = r1 * r2;
  return r1;
}

rational& operator /= (rational& r1, rational r2) {
  r1 = r1 / r2;
  return r1;
}

int operator == (rational& r1, rational& r2) {
  // Zahlen sind immer normiert!
  return r1.p == r2.p && r1.q == r2.q;
}

int operator != (rational& r1, rational& r2) {
  return ! (r1 == r2);
}

ostream& operator << (ostream& s, rational r) {
  return s << r.p << "/" << r.q;
}
```

Die Gefahr eines Überlaufs bei der `long`-Arithmetik läßt sich verringern, wenn wir bei einigen Operatoren zusätzliche Kürzungen versuchen. Hier sind die Alternativen:

```
rational operator + (rational r1, rational r2) {
  rational r;
  long t = ggT (r1.q, r2.q);
  r.q = r1.q / t;
  r.p = r.q * r2.p + (r2.q / t) * r1.p;
  r.q *= r2.q;
  return rational (r.p, r.q); // kürzen!
}

rational operator * (rational r1, rational r2) {
  long t = ggT (r1.p, r2.q);
  if (t > 1) {r1.p /= t; r2.q /= t;}
  t = ggT (r1.q, r2.p);
  if (t > 1) {r1.q /= t; r2.p /= t;}
  return rational (r1.p * r2.p, r1.q * r2.q);
}

rational operator / (rational r1, rational r2) {
  long t = ggT (r1.p, r2.p);
  if (t > 1) {r1.p /= t; r2.p /= t;}
  t = ggT (r1.q, r2.q);
  if (t > 1) {r1.q /= t; r2.q /= t;}
  return rational (r1.p * r2.q, r1.q * r2.p);
}
```

Es folgt ein kleines Programm, das einige der Operatoren testet:

```
////////////////////////// RATIOTST.CPP ////////////////////////////
//                                                                //
//    Test von RATIONAL.CPP (Arithmetik mit rationalen Zahlen)    //
//                                                                //
////////////////////////////////////////////////////////////////////
//
// Projekt besteht aus RATIOTST.CPP, RATIONAL.H, RATIONAL.CPP
//
#include "rational.h"

void main (void) {
  cout << "\n\n";
  for (int i=10; i<15; i++)
    for (int j=11; j<15; j++) {
      rational r1 (15,i);
      rational r2 (10,j);
      rational d = r1 - r2;
      rational p = r1 * r2;
      rational q = r1 / r2;
      cout << "r1=" << r1 << ", r2=" << r2 << ", r1-r2=" << d
           << ", r1*r2="  << p  <<  ", r1/r2=" << q << "\n";
    }
}
```

Dieses Programm erzeugt folgende Ausgabe:

```
r1=3/2, r2=10/11, r1-r2=13/22, r1*r2=15/11, r1/r2=33/20
r1=3/2, r2=5/6, r1-r2=2/3, r1*r2=5/4, r1/r2=9/5
r1=3/2, r2=10/13, r1-r2=19/26, r1*r2=15/13, r1/r2=39/20
r1=3/2, r2=5/7, r1-r2=11/14, r1*r2=15/14, r1/r2=21/10
r1=15/11, r2=10/11, r1-r2=5/11, r1*r2=150/121, r1/r2=3/2
r1=15/11, r2=5/6, r1-r2=35/66, r1*r2=25/22, r1/r2=18/11
r1=15/11, r2=10/13, r1-r2=85/143, r1*r2=150/143, r1/r2=39/22
r1=15/11, r2=5/7, r1-r2=50/77, r1*r2=75/77, r1/r2=21/11
r1=5/4, r2=10/11, r1-r2=15/44, r1*r2=25/22, r1/r2=11/8
r1=5/4, r2=5/6, r1-r2=5/12, r1*r2=25/24, r1/r2=3/2
r1=5/4, r2=10/13, r1-r2=25/52, r1*r2=25/26, r1/r2=13/8
r1=5/4, r2=5/7, r1-r2=15/28, r1*r2=25/28, r1/r2=7/4
r1=15/13, r2=10/11, r1-r2=35/143, r1*r2=150/143, r1/r2=33/26
r1=15/13, r2=5/6, r1-r2=25/78, r1*r2=25/26, r1/r2=18/13
r1=15/13, r2=10/13, r1-r2=5/13, r1*r2=150/169, r1/r2=3/2
r1=15/13, r2=5/7, r1-r2=40/91, r1*r2=75/91, r1/r2=21/13
r1=15/14, r2=10/11, r1-r2=25/154, r1*r2=75/77, r1/r2=33/28
r1=15/14, r2=5/6, r1-r2=5/21, r1*r2=25/28, r1/r2=9/7
r1=15/14, r2=10/13, r1-r2=55/182, r1*r2=75/91, r1/r2=39/28
r1=15/14, r2=5/7, r1-r2=5/14, r1*r2=75/98, r1/r2=3/2
```

Datumsberechnungen

Als weiteres Beispiel für das Überladen von Operatoren betrachten wir eine Klasse `Datum` für Berechnungen mit Kalenderdaten. Zur Berechnung der Zeitspanne in Tagen zwischen zwei Kalenderdaten bietet sich der Minus-Operator an, zur Erhöhung eines Kalenderdatums um eine Anzahl von Tagen der Operator +=. Intern wird zu jedem Datum eine fortlaufende Tageszahl berechnet. Dabei ist es gleichgültig, auf welches Anfangsdatum sich diese Zahl bezieht, weil sie nur für die Berechnung von Datumsdifferenzen verwendet wird. Weiter sollen Wochentage und bewegliche Feste

berechnet werden und Kalenderdaten im Klartext angegeben werden können. So kommen wir zur folgenden Header-Datei:

```
////////////////////////////////////////////////////////////////////
//        DATUM.H  -  Klasse Datum für Kalenderberechnungen       //
////////////////////////////////////////////////////////////////////

#ifndef DATUM_H
#define DATUM_H

#ifndef BOOLEAN
#define BOOLEAN
enum {false, true};
typedef int boolean;
#endif BOOLEAN

class Datum {
public:
  Datum (int Tag=1, int Monat=1, int Jahr=2000);
  void heute ();                   // übernimmt das Datum von MS-DOS
  int Wochentag ();                // 1=Mo, 2=Di, 3=Mi, ..., 7=So
  char *String ();                 // z.B. "Mo, 31.12.1990"
  int Tag ();
  int Monat ();
  int Jahr ();
  void Ostern (int Jahr);
  void Busstag (int Jahr);
  void operator += (long Tage);
  void operator -= (long Tage);
  friend long operator - (Datum, Datum); // Differenz in Tagen
protected:
  int t, m, j;                     // Tag, Monat, Jahr
  long Tagnummer;                  // Laufende Nummer
  void berechneTagnummer ();       // aus t, m, j
  void berechneTMJ ();             // aus Tagnummer
};

extern char *MoName[12];  // Monatsnamen
extern char WoTag[7][3];  // Namen der Wochentage (2 Buchstaben)

extern int Monatslaenge (int Monat, int Jahr=1);
extern boolean Schaltjahr (int Jahr);

///////////////////////// inline-Funktionen: ///////////////////////

inline int Datum::Tag ()   {return t;}
inline int Datum::Monat () {return m;}
inline int Datum::Jahr ()  {return j;}

#endif DATUM_H
```

Das folgende kurze Beispielprogramm demonstriert einige typische Methoden der Klasse Datum:

```
////////////////////////////////////////////////////////////////////
//        DAT-TEST.CPP  -  Demonstration der Klasse Datum         //
////////////////////////////////////////////////////////////////////

#include <iostream.h>
#include "datum.h"

main () {
  Datum Mozart (27,1,1756), jetzt, Fest;
  jetzt.heute ();
  int J = jetzt.Jahr ();
```

```
  cout << "\nBewegliche Feste " << J << "\n\n";
  Fest.Ostern (J);
  cout << "Ostersonntag:    " << Fest.String()+4 << "\n";
  Fest += 49;
  cout << "Pfingstsonntag:  " << Fest.String()+4 << "\n";
  Fest.Busstag (J);
  cout << "Buß- und Bettag: " << Fest.String()+4 << "\n\n";
  cout << "Heute ist " << jetzt.String() << ".\n";
  cout << "Mozart ist am " << Mozart.String() << " geboren.\n";
  cout << "Seitdem sind " << jetzt-Mozart << " Tage vergangen.\n";
}
```

Es folgt die Implementierung der Klasse `Datum`. Um innerhalb eines Jahres nicht zwischen Tagen vor und nach dem Schalttag unterscheiden zu müssen, rechnen wir hier intern so, als ob das Jahr erst am 1. März anfinge (So war es tatsächlich vor der Julianischen Kalenderreform!). Weiter sei zum Verständnis daran erinnert, daß (seit der Gregorianischen Kalenderreform) die durch 100, aber nicht durch 400 teilbaren Jahre keine Schaltjahre sind (zum Beispiel war 1900 kein Schaltjahr, 2000 ist aber ein Schaltjahr).

```
////////////////////////////////////////////////////////////////////
//       DATUM.CPP  -  Klasse Datum für Kalenderberechnungen      //
////////////////////////////////////////////////////////////////////

#include <string.h>
#include <stdlib.h>
#include <dos.h>    // verwendet: MK_FP, intdos
#include "datum.h"

void DosGetDate (int& Tag, int& Monat, int& Jahr) {
  REGS  Reg;
  Reg.h.ah = 0x2A;        // MS-DOS 0x2A: Get Date
  intdos (&Reg, &Reg);
  Tag   = Reg.h.dl;
  Monat = Reg.h.dh;
  Jahr  = Reg.x.cx;
}

char *MoName[12] = {
  "Januar","Februar","März","April","Mai","Juni",
  "Juli","August","September","Oktober","November","Dezember"
};

char WoTag[7][3] = {"Mo","Di","Mi","Do","Fr","Sa","So"};

static int MoAnf[12] = {
     0,  31, 61, 92, 122, 153, 184, 214, 245, 275, 306, 337};
  //Mrz Apr Mai Jun  Jul  Aug  Sep  Okt  Nov  Dez  Jan  Feb

Datum::Datum (int Tag, int Monat, int Jahr) {
  t = Tag;
  m = Monat;
  j = Jahr;
  berechneTagnummer ();
}

void Datum::heute() {
  int t,m,j;
  DosGetDate (t,m,j);
  *this = Datum (t,m,j);
}

int Datum::Wochentag () {
  return 1 + (Tagnummer+2) % 7;
```

```
}

char *Datum::String () {
  static char s[15];                 // Beispiel: 01.07.1991
  strcpy (s, WoTag[Wochentag()-1]);  // "Di"
  s[2] = ',';                        // "Di,???????????"
  itoa (t + 100, s+3, 10);           // "Di,101"
  s[3] = ' ';                        // "Di, 01"
  itoa (m + 100, s+6, 10);           // "Di, 01107"
  s[6] = s[9] = '.';                 // "Di, 01.07.????"
  itoa (j, s+10, 10);                // "Di, 01.07.1991"
  return s;
}

void Datum::Ostern (int Jahr) { // Formel von Gauß/Hartmann
                        // nach "Schlag nach!",
  int q = Jahr / 4;     // Bibliographisches Institut, Mannheim 1960
  int a = Jahr % 19;    // gilt in dieser Form nur von 1900...2099
  int b = (204 - 11 * a) % 30;
  if (b > 27) b--;
  int c = (Jahr + q + b - 13) % 7;
  int t = 28 + b - c;
  if (t > 31) *this = Datum (t-31, 4, Jahr);
  else *this = Datum (t, 3, Jahr);
}

void Datum::Busstag (int Jahr) {
  // Bußtag ist Mittwoch, 16. bis 22. November
  *this = Datum (22, 11, Jahr);
  *this -= (Wochentag() + 4) % 7;
}

void Datum::operator += (long Tage) {
  Tagnummer += Tage;
  berechneTMJ ();
}

void Datum::operator -= (long Tage) {
  Tagnummer -= Tage;
  berechneTMJ ();
}

long operator - (Datum D1, Datum D2) {
  return D1.Tagnummer - D2.Tagnummer;
}

static long Maerztag (int j) {
  // berechnet fortlaufende Tagesnummer des 0. März im Jahr j
  // (Basis: 0. März 0 steht für 28.Februar 1 v.Chr.)
  return 365L * j          // Länge der Gemeinjahre
            + j / 4        // Alle 4 Jahre ist Schaltjahr ...
            - j / 100      // außer in den durch 100 ...
            + j / 400;     // aber nicht durch 400 teilbaren Jahren
}

void Datum::berechneTagnummer () {
  int M = m, J = j;
  if (M > 2) M -= 3;          // März = 0, April = 1 ...
  else {M +=9; J--;}          // Jan. u. Feb. zum alten Jahr
  Tagnummer = Maerztag (J)    // Tagnummer des 1.März
            + MoAnf[M]        // Länge der verstrichenen Monate ab März
            + t - 1;          // Tage im laufenden Monat
}

void Datum::berechneTMJ () {
  int Rest;
  j = 1 + Tagnummer / 365;      // j ist mindestens um 1 zu groß
  do {
    j--;                        // römisches Jahr suchen
```

```
    Rest = Tagnummer - Maerztag (j);
  }
  while (Rest < 0);            // Rest = verstrichene Tage ab 1. März
  m = 11;                      // Anfang mit letzem Monat (Februar)
  while (MoAnf[m] > Rest)
    m--;                       // richtigen Monat suchen
  t = Rest + 1 - MoAnf[m];
  if (m > 9) {m -= 9; j++;}    // in "richtige" Monate umrechnen
  else m += 3;
}

int Monatslaenge (int Monat, int Jahr) {
  static int MoLg[12] = {
    31,28,31,30,31,30,31,31,30,31,30,31};
  if (Monat == 2) return Schaltjahr (Jahr) ? 29 : 28;
  else return MoLg[Monat-1];
}

boolean Schaltjahr (int Jahr) {
  return (Jahr % 4 == 0)
    && ((Jahr % 100 != 0) || (Jahr % 400 == 0));
}
```

Auf der Diskette zum Buch finden Sie ein Programm KALENDER, das einen Jahreskalender einschließlich der beweglichen Feste auf dem Bildschirm zeigt. Zunächst wird das aktuelle Jahr dargestellt. Mit den Cursortasten können beliebige Jahre gewählt werden.

3.6 Definition von Typumwandlungen

Ganze Zahlen sind spezielle rationale Zahlen. Es ist daher natürlich, arithmetische Operationen zwischen ganzen und rationalen Zahlen zu erlauben. Wenn wir alle möglichen Kombinationen berücksichtigen wollen, müssen wir die Klassendeklaration allein für die Addition so erweitern:

```
class rational {
public:
  ...
  friend rational operator + (rational r1, rational r2);
  friend rational operator + (int i, rational r2);
  friend rational operator + (rational r1, int i);
 ...
};
```

Sicher sind Sie nicht begeistert von der Vorstellung, für alle Operatoren alle möglichen Typ-Kombinationen zu implementieren. Zum Glück geht es auch einfacher: Es genügt, die Addition zwischen zwei Operanden vom Typ `rational` zu definieren, wenn der Compiler weiß, wie er `int`-Zahlen in den Typ `rational` umwandeln kann.

Bei den Standard-Datentypen haben Sie die impliziten Typ-Konvertierungen bereits kennengelernt. Aufgrund dieser automatischen Umwandlungen dürfen Sie zum Beispiel der Bibliotheksfunktion `sin` Argumente eines beliebigen Zahltyps übergeben.

Jetzt wollen wir sehen, wie man implizite Typ-Umwandlungen bei eigenen Typen selbst definieren kann. Wir unterscheiden zwei Fälle:

Umwandlung von einem beliebigen Typ in einen Klassentyp

Hierzu muß in der Klasse ein Konstruktor mit einem Argument des beliebigen Typs definiert werden. Der Konstruktor der Klasse `rational` ist ein Beispiel hierfür:

```
  rational (long Zaehler = 0, long Nenner = 1);
```

Hiermit werden eigentlich drei Konstruktoren deklariert: ohne Argument, mit einem und mit zwei Argumenten. Hier interessiert der Konstruktor mit einem Argument. Er ermöglicht zum Beispiel die folgende Zuweisung:

```
rational r;
long     i;
.....
r = i;  // Konvertierung von long nach rational
```

Umwandlung von einem Klassentyp in einen beliebigen Typ

Hierfür kann das Schlüsselwort `operator` in einer neuen Bedeutung verwendet werden: Es wird eine Elementfunktion der Form `operator` **`Typ`** `();` deklariert.

Als Beispiel fügen wir in die Klasse `rational` eine implizite Typ-Umwandlung in den Typ `double` ein. Dazu setzen wir im `public`-Abschnitt der Klassen-Deklaration die zusätzliche Zeile

```
  operator double ();
```

ein. Die Implementierung sieht so aus:

```
rational::operator double () {
  return double(p) / double (q);
}
```

Es folgt ein etwas komplizierteres, kommentiertes Skelett-Beispiel, das beide Umwandlungsarten und mögliche Fehlerquellen aufzeigt. Hier ist die Umwandlung von `TypA` nach `TypB` mit beiden Methoden definert. Der Compiler meldet diese Mehrdeutigkeit als Fehler (aber nur, wenn im Programm wirklich eine Umwandlung benötigt wird):

```
// Demonstration von Typ-Umwandlungen:
class TypB;

class TypA {
public:
  TypA ();
  TypA (int i);       // Konstruktor zur Umwandlung int --> TypA
  operator int ();    // Umwandlung TypA --> int
  operator TypB ();   // Umwandlung TypA --> TypB
private:
  int n;
};

class TypB {
public:
  TypB ();
```

```
  TypB (TypA a);      // Konstruktor zur Umwandlung TypA --> TypB
  operator TypA ();  // Umwandlung TypB --> TypA
private:
  int n;
};

TypA::TypA ()             {}
TypA::operator int ()     {return n;}
TypA::TypA (int i)        {n = i;}

TypB::TypB ()             {}
TypB::TypB (TypA a)       {n = a;}
TypB::operator TypA ()    {return TypA (n);}

int main () {
  TypA a;
  TypB b;
  int i;
  i = a;
  i = b;  // Fehler: keine Umwandlung von TypB nach int definiert
  a = i;
  a = b;
  b = i;
  b = a;  // Fehler: Typumwandlung mehrdeutig:
          // (a) TypA::operator TypB
          // (b) TypB::TypB (TypA)
}
```

Übungsaufgabe 11:

Erweitern Sie die Klasse `bitset` im Programm `BITSET3.CPP` um implizite Typ-Umwandlungen von und nach `int`.

Übungsaufgabe12:

Entfernen Sie aus der Klasse `rational` die Operatoren `==` und `!=`. Schreiben Sie dann ein Testprogramm, in dem zwei rationale Zahlen auf Gleichheit geprüft werden. Warum meldet der Compiler keinen Fehler, obwohl die Klasse keinen Gleichheitsoperator mehr hat, und was ist der Unterschied in der Behandlung der Gleichheit?

Beispiel: Auf eine Datei ausgelagerte große Vektoren

Das folgende Beispiel gehört eigentlich zum Thema *Überladen von Operatoren*, zeigt aber auch eine Anwendung der Typumwandlungen. Wir wollen einen Typ `fVektor` definieren, der wie ein Pascal-Array einen beliebigen Indexbereich hat. Dazu müssen wir die Indizes, die in C++ immer ab Null gezählt werden, umrechnen. Weiter wollen wir annehmen, daß der Hauptspeicher zu klein ist, um den Vektor unterzubringen. Wir lagern den Vektor daher auf eine Datei aus. Trotz allem soll er in der üblichen Weise mit dem Operator `[]` indiziert werden.

Eine naheliegende Möglichkeit wäre es nun, den Operator `[]` so zu überladen, daß er den Wert des angesprochenen Elements zurückliefert. Damit wären zwar Anweisungen wie

```
n = A[i];
```

möglich, nicht aber eine Zuweisung der Form

```
A[i] = n;
```

Offenbar darf der Operator `[]` nicht den Wert des Objekts (R-Wert), sondern eine Referenz (L-Wert) zurückliefern. Aber auch das löst unser Problem noch nicht, denn eine Referenz steht für ein Objekt im Hauptspeicher, unsere Objekte befinden sich aber an einer bestimmten Stelle einer Datei. Deshalb definieren wir einen Typ `DateiPos`, der für eine bestimmte Position in einer geöffneten Datei steht. Jetzt bekommen wir die Zuweisung in den Griff, indem wir eine Typumwandlung von `DateiPos` nach `int` definieren. Um die zweite Zuweisung zu ermöglichen, überladen wir den Operator = für die Zuweisung von `int` nach `DateiPos`. Jetzt läuft die Zuweisung

```
A[i] = n;
```

so ab: `A[i]` ist vom Typ `DateiPos`. Dieser wird der Wert `n` zugewiesen. Das hat die Wirkung, daß an der richtigen Stelle in der Datei der Wert von `n` geschrieben wird.

Die nun folgende Implementierung soll nur das Prinzip darstellen und verzichtet bewußt auf eine Fehlerbehandlung.

```
//////////////////////////// FVEKTOR.CPP ///////////////////////////
//                                                                //
//    Auf Datei ausgelagerte große Vektoren (Prinzip-Version)     //
//                                                                //
////////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <string.h>
#include <iostream.h>
#include <dir.h>       // MAXPATH

#define INT int        // zur einfachen Umstellung auf andere Typen

/////////////////// Klassen DateiPos und fVektor /////////////////
//                                                                //
//  fVektor definiert einen Vektor, dessen Inhalt in einer Datei  //
//  gehalten wird.                                                //
//  DateiPos dient dem Verweis auf eine Dateiposition.            //
//                                                                //
////////////////////////////////////////////////////////////////////

class DateiPos {
public:
  operator INT ();            // Typkonvertierung
  void operator = (INT i);
  friend ostream& operator << (ostream& s, DateiPos p);
protected:
  friend class fVektor;
  FILE   *Dateizeiger;
  long   Position;
};

class fVektor {
public:
  fVektor (long i0, long i1, char *Datei);
  ~fVektor ();
  DateiPos operator[] (long i);
protected:
```

```
  FILE *Dateizeiger;
  char Dateiname[MAXPATH];
  long Groesse, Basis;
};

///////////////////////// Implementierung: /////////////////////////

DateiPos::operator INT () {           // Typkonvertierung
  INT n;
  fseek (Dateizeiger, Position, SEEK_SET);
  fread (&n, sizeof(INT), 1, Dateizeiger);
  return n;
}

void DateiPos::operator = (INT i) {
  fseek (Dateizeiger, Position, SEEK_SET);
  fwrite (&i, sizeof(INT), 1, Dateizeiger);
}

ostream& operator << (ostream& s, DateiPos p) {
  return s << INT (p);
}
```

Dem Konstruktor wird der Indexbereich und der Name der temporären Datei angegeben:

```
fVektor::fVektor (long i0, long i1, char *Datei) {
  Groesse = i1 - i0 + 1;
  Basis = i0;
  strcpy (Dateiname, Datei);
  Dateizeiger = fopen (Datei, "w+");
}
```

Der Destruktor löscht die Datei automatisch:

```
fVektor::~fVektor () {
  fclose (Dateizeiger);
  remove (Dateiname);
}

DateiPos fVektor::operator[] (long i) {
  DateiPos p;
  p.Dateizeiger = Dateizeiger;
  p.Position = (i - Basis) * sizeof(INT);
  return p;
}

//////////////////////////////// Test: /////////////////////////////

main () {
  int i;
  fVektor A (10000, 20000, "C:\\$VEKTOR$.TMP");
  cout << "\n";
  for (i=10000; i<20000; i+=1000) cout << A[i] << " ";
  cout << "\n";
  for (i=10000; i<20000; i++) A[i] = i;
  for (i=10000; i<20000; i+=1000) cout << A[i] << " ";
  cout << "\n";
}
```

3.7 Statische Elemente

Objekte eines Klassentyps greifen auf gemeinsame Klassenmethoden zu. Sie können sich

nur in der aktuellen Belegung ihrer Datenelemente unterscheiden. Manchmal werden auch Datenelemente benötigt, die nur einmal für die Klasse existieren und von allen Objekten genutzt werden können, zum Beispiel, um die zum Erzeugen von Objekten zur Verfügung stehenden Ressourcen zu überwachen. Hierfür wird das Schlüsselwort `static` in einer neuen Bedeutung verwendet:

- Ein mit deklariertes *Datenelement* existiert nur einmal pro Klasse. Es kann von allen Objekten angesprochen werden.
- Eine mit deklarierte *Elementfunktion* `f()` der Klasse `X` wird mit `X::f()` aufgerufen. Sie kann alternativ auch von einem Datenelement `x` der Klasse mit `x.f()` aufgerufen werden, hat aber auch dann keinen `this`-Zeiger und darf demnach nicht auf nicht-statische Datenelemente zugreifen.

Ein Skelett-Programm zeigt die Möglichkeiten und Grenzen auf:

```
class A {
public:
  int i;
  static int j;       // statisches Datenelement
  void f ();
  static void g ();   // statische Elementfunktion
};

void A::f () {     // Eine nicht-statische Elementfunktion
  i++;             // darf auf nicht-statische
  j++;             // und statische Datenelemente zugreifen
}

void A::g () {
  i++;           // Fehler: statische Funktion kann nicht
                 // auf nicht-statisches Element zugreifen
  j++;
}

int A::j = 0;    // so werden statische Detenelemente initialisiert

int main () {
  A a;
  a.f ();
  A::f ();       // Fehler: nicht-statische Elementfunktion kann nur von
                 // einem Objekt, nicht von der Klasse aufgerufen werden
  a.g ();        // vorher muß A::g () korrigiert werden!
  A::g();
}
```

3.7a Zeiger auf Elementfunktionen

[wird noch nachgetragen]

3.8 Vererbung

Mit Hilfe der Vererbung ist es möglich, Klassen zu definieren, die sich von einer schon vorhandenen Klasse nur in einigen Details unterscheiden. Damit wird die Wiederverwendung von Klassen enorm erleichtert.

Ein einfaches Grafik-Programm

Zur Motivation der Vererbung betrachten wir einen primitiven Prototyp eines CAD-Programms. Mit diesem einfachen Programm können Sie lediglich zwei vorgegebene Objekte, einen Kreis und ein Rechteck, auf dem Bildschirm bewegen.

Dazu werden zwei Klassen, `Kreis` und `Rechteck`, definiert, die beide neben ihrem Konstruktor die folgenden Methoden kennen:

```
void zeige();     // zeichnet das Objekt in Vordergrundfarbe
void loesche();   // zeichnet das Objekt in Hintergrundfarbe
void bewege (int dx, int dy); // relative Verschiebung
```

Die Methode `bewege` läuft in drei Schritten ab:

- 1. Das Objekt wird gelöscht
- 2. Die Koordinaten werden verschoben
- 3. Das Objekt wird mit den neuen Koordinaten gezeigt

Diese Art der Bewegung auf dem Bildschirm hat ihre Grenzen. Sie können das leicht feststellen, wenn Sie ein Objekt über den Hilfetext oben auf dem Bildschirm bewegen. Beim Löschen wird einfach schwarz gezeichnet, ohne Rücksicht auf den Hintergrund. Besser wäre es natürlich, vor jeder Bewegung den Bildhintergrund zu speichern und später wieder zu rekonstruieren. Sie sollen hier aber nicht durch technische Details vom prinzipiellen Problem abgelenkt werden.

Im Hauptprogramm werden je ein Kreis und ein Rechteck erzeugt. Der Kreis wird als aktuelles Objekt angesehen. Das aktuelle Objekt (der Kreis) wird an seiner Anfangsposition auf dem Bildschirm gezeigt. Dann wird die Tastatur abgefragt, bis die `Esc`-Taste gedrückt wird. Dabei werden die Tastendrücke so verarbeitet:

Taste `'K'`: Der Kreis wird gezeigt

Taste `'R'`: Das Rechteck wird gezeigt

Cursortasten: Das aktuelle Objekt wird in Pfeilrichtung verschoben

Zur grafischen Darstellung werden die in GRAPHICS.H deklarierten Routinen (BGI-Grafik) verwendet. Die Implementierung enthält keine besonderen Tricks oder Schwierigkeiten:

```
////////////////////////////////////////////////////////////////////
//                         FIGUREN0.CPP                          //
//            Geometrische Objekte ohne Vererbung                 //
////////////////////////////////////////////////////////////////////

#include <graphics.h>
#include <ctype.h>
#include <conio.h>

class Kreis {
public:
  Kreis(int x0, int y0, int r0);
  void zeige();
```

```
  void loesche();
  void bewege (int dx, int dy);
private:
  int x;
  int y;
  int Radius;
};

class Rechteck {
public:
  Rechteck (int x0, int y0, int x1, int y1);
  void zeige();
  void loesche();
  void bewege (int dx, int dy);
private:
  int x;
  int y;
  int a, b;  // die beiden Seiten
};

//////////////// Implementierung der Klasse Kreis: ///////////////

Kreis::Kreis(int x0, int y0, int r0) {
  x = x0;
  y = y0;
  Radius = r0;
}

void Kreis::zeige() {
  circle(x, y, Radius);
}

void Kreis::loesche() {
  setcolor (BLACK);
  circle(x, y, Radius);
  setcolor (WHITE);
}

void Kreis::bewege (int dx, int dy) {
  loesche ();
  x += dx;
  y += dy;
  zeige ();
}

/////////////// Implementierung der Klasse Rechteck: /////////////

Rechteck::Rechteck (int x0, int y0, int x1, int y1) {
  x = x0;
  y = y0;
  a = x1 - x0;
  b = y1 - y0;
}

void Rechteck::zeige() {
  rectangle (x, y, x+a, y+b);
}

void Rechteck::loesche() {
  setcolor (BLACK);
  rectangle (x, y, x+a, y+b);
  setcolor (WHITE);
}

void Rechteck::bewege (int dx, int dy) {
  loesche ();
  x += dx;
  y += dy;
  zeige ();
```

```
}

//////////////////////// Test der Klassen: ///////////////////////

int main() {
  char c;
  int graphdriver = DETECT, graphmode;
  int dx, dy;
  initgraph (&graphdriver, &graphmode, "\\borlandc\\bgi");
  outtextxy (10, 8, "Figur mit Cursortasten bewegen. ");
  outtextxy (10,24, "Figur wechseln: [R]echteck  [K]reis");
  outtextxy (10,40, "Ende: [Esc]");
  Kreis K (getmaxx()/2, getmaxy()/2, 40);
  Rechteck R (0, 0, 80, 60);
  char aktuelleFigur = 'K';
  K.zeige();
  do {
    c = getch();      // auf Tastendruck warten
    if (c != 0) {     // normale Zeichentaste: andere Figur?
      switch (aktuelleFigur) {
        case 'K': K.loesche (); break;
        case 'R': R.loesche (); break;
      }
      switch (toupper (c)) {
        case 'K': aktuelleFigur = 'K'; break;
        case 'R': aktuelleFigur = 'R'; break;
      }
      switch (aktuelleFigur) {
        case 'K': K.zeige (); break;
        case 'R': R.zeige (); break;
      }
    }
    else {            // erweiterter Tastencode: Cursorbewegung?
      c = getch();
      switch (c) {
        case 72: dx = 0; dy = -10; break; // aufwärts
        case 80: dx = 0; dy =  10; break; // abwärts
        case 75: dx = -10; dy = 0; break; // links
        case 77: dx = 10; dy = 0;  break; // rechts
      }
      switch (aktuelleFigur) {
        case 'K': K.bewege (dx, dy); break;
        case 'R': R.bewege (dx, dy); break;
      }
    }
  } while (c != '\33');  // Esc-Taste
  closegraph();
  return 0;
}
```

Zusammenfassung von Gemeinsamkeiten: Vererbung

Wenn Sie die beiden Klassen `Kreis` und `Rechteck` betrachten, sehen Sie, daß sie folgende gemeinsamen Elemente besitzen:

- - Die Methoden `zeige`, `loesche` und `bewege`,
- - die Datenelemente `x` und `y`.

Die Methoden unterscheiden sich allerdings in ihrer Implementierung. In der Wissenschaft, wie im täglichen Leben, versucht man der Fülle von Objekten und Phänomenen Herr zu werden, indem man das Gemeinsame herausarbeitet und so Klassen bildet, deren Unterabteilungen sich in möglichst wenig Details abheben. Ein bekanntes

Beispiel für solche Klassenbildungen ist die von Carl von Linné begründete Systematik der Tier- und Pflanzenarten. So gehört zum Beispiel ein Buntspecht der Ordnung der Spechte an, die Spechte der Klasse der Vögel und die Vögel dem Unterstamm der Wirbeltiere. Die Färbung des Buntspechts ist ein Artmerkmal. Seinen kräftigen Meißelschnabel hat er mit allen Spechten gemeinsam, das Federkleid mit allen Vögeln und das Knochengerüst mit allen Wirbeltieren.

Auch in der Programmierung hat man es immer wieder mit Datentypen zu tun, die sich zwar nicht gleichen, aber doch gemeinsame Elemente besitzen. Um hier hierarchische Modelle zu ermöglichen, besitzen objektorientierte Sprachen die Möglichkeit der *Vererbung*.

Die Gemeinsamkeiten zwischen Buntspechten und Grünspechten führen zur Bildung der Oberklasse Specht. Welche Gemeinsamkeiten besitzen unsere Klassen `Kreis` und `Rechteck`? Beide besitzen einen Bezugspunkt (x, y), der ihre aktuelle Lage beschreibt. Wir definieren also eine Klasse `Punkt`:

```
class Punkt {
public:
  Punkt (int x0, int y0);
private:
  int x;
  int y;
};
```

mit dem Konstruktor

```
Punkt::Punkt(int x0, int y0)  {
  x = x0;  y = y0;
}
```

Jetzt wollen wir erklären, daß ein Kreis eine von Punkt abgeleitete Klasse ist. Das heißt, `Kreis` besitzt zunächst einmal alle Eigenschaften von `Punkt`. In der Definition von `Kreis` werden nur die Eigenschaften angegeben, in denen sich ein `Kreis` von einem `Punkt` abhebt:

```
class Kreis: public Punkt { // Kreis ist von Punkt abgeleitet
public:
  Kreis(int x0, int y0, int r0);
  void zeige();
  void loesche();
  void bewege (int dx, int dy);
private:
  int Radius;
};
```

Die Implementierung können wir von der ersten Version übernehmen. Es folgt das vollständige Programm.

```
////////////////////////////////////////////////////////////////////
//                         FIGUREN1.CPP                           //
//              Geometrische Objekte mit Vererbung                //
////////////////////////////////////////////////////////////////////
#include <graphics.h>
#include <ctype.h>
```

```
#include <conio.h>

class Punkt {
public:
  Punkt (int x0, int y0);
  int x;  // Die Datenkomponenten sind hier vorläufig public
  int y;  // Der Grund wird im nächsten Abschnitt verständlich!
};

class Kreis: public Punkt {
public:
  Kreis(int x0, int y0, int r0);
  void zeige();
  void loesche();
  void bewege (int dx, int dy);
private:
  int Radius;
};

class Rechteck: public Punkt {
public:
  Rechteck (int x0, int y0, int x1, int y1);
  void zeige();
  void loesche();
  void bewege (int dx, int dy);
private:
  int a, b;  // die beiden Seiten
};

///////////////// Implementierung der Klasse Punkt: ////////////////

Punkt::Punkt(int x0, int y0)  {
  x = x0;  y = y0;
}

///////////////// Implementierung der Klasse Kreis: ////////////////

Kreis::Kreis(int x0, int y0, int r0): Punkt (x0, y0) {
  Radius = r0;
}

void Kreis::zeige() {
  circle(x, y, Radius);
}

void Kreis::loesche() {
  setcolor (BLACK);
  circle(x, y, Radius);
  setcolor (WHITE);
}

void Kreis::bewege (int dx, int dy) {
  loesche ();
  x += dx;
  y += dy;
  zeige ();
}

//////////////// Implementierung der Klasse Rechteck: //////////////

Rechteck::Rechteck (int x0, int y0, int x1, int y1): Punkt (x0,y0) {
  x = x0;
  y = y0;
  a = x1 - x0;
  b = y1 - y0;
}

void Rechteck::zeige() {
  rectangle (x, y, x+a, y+b);
```

```
}

void Rechteck::loesche() {
  setcolor (BLACK);
  rectangle (x, y, x+a, y+b);
  setcolor (WHITE);
}

void Rechteck::bewege (int dx, int dy) {
  loesche ();
  x += dx;
  y += dy;
  zeige ();
}

///////////////////////// Test der Klassen: ///////////////////////

int main() {
  char c;
  int graphdriver = DETECT, graphmode;
  initgraph (&graphdriver, &graphmode, "\\borlandc\\bgi");
  outtextxy (10, 8, "Figur mit Cursortasten bewegen. ");
  outtextxy (10,24, "Figur wechseln: [R]echteck  [K]reis");
  outtextxy (10,40, "Ende: [Esc]");
  Kreis K (getmaxx()/2, getmaxy()/2, 40);
  Rechteck R (0, 0, 80, 60);
  char aktuelleFigur = 'K';
  K.zeige();
  do {
    c = getch();      // auf Tastendruck warten
    if (c != 0) {     // normale Zeichentaste: andere Figur?
      switch (aktuelleFigur) {
        case 'K': K.loesche (); break;
        case 'R': R.loesche (); break;
      }
      switch (toupper (c)) {
        case 'K': aktuelleFigur = 'K'; break;
        case 'R': aktuelleFigur = 'R'; break;
      }
      switch (aktuelleFigur) {
        case 'K': K.zeige (); break;
        case 'R': R.zeige (); break;
      }
    }
    else {            // erweiterter Tastencode: Cursorbewegung?
      c = getch();
      switch (aktuelleFigur) {
        case 'K':
          switch (c) {
            case 72: K.bewege (0, -10); break; // aufwärts
            case 80: K.bewege (0,  10); break; // abwärts
            case 75: K.bewege (-10, 0); break; // links
            case 77: K.bewege (10,  0); break; // rechts
          }
          break;
        case 'R':
          switch (c) {
            case 72: R.bewege (0, -10); break; // aufwärts
            case 80: R.bewege (0,  10); break; // abwärts
            case 75: R.bewege (-10, 0); break; // links
            case 77: R.bewege (10,  0); break; // rechts
          }
          break;
      }
    }
  } while (c != '\33');  // Esc-Taste
  closegraph();
  return 0;
}
```

Vererbung des Zugriffsschutzes

Es bleibt nur noch ein kleines Problem: Die Elemente `x` und `y` der Klasse `Punkt` sind als `private` deklariert. Deshalb kann die Klasse `Kreis` darauf nicht zugreifen. Um private Elemente auch abgeleiteten Klassen zugänglich zu machen, verwendet man das Schlüsselwort `protected`. Während `privat`-Elemente nur Elementfunktionen der Klasse selbst zugänglich sind, können `protected`-Elemente auch von abgeleiteten Klassen angesprochen werden.

Der Vollständigkeit halber sei hier auch die Bedeutung des Schlüsselworts `public` im Kopf abgeleiteter Klassen erwähnt. In der Definition

```
class B: public A {...};
```

erklärt der *Zugriffsmodifizierer* `public`, daß die Zugriffsattribute bei abgeleiteten Klassen erhalten bleiben. Wenn Sie den Zugriffsmodifizierer weglassen (oder `private` verwenden), sind in der abgeleiteten Klasse alle aus der Basisklasse geerbten Elemente privat.

Einen genauen Überblick soll die folgende Tabelle geben:

Zugriff in abgeleiteter Klasse bei:	Zugriffsmodifizierer	
Zugriff in Basisklasse	`public`	`private`
`public`	`public`	`private`
`protected`	`protected`	`private`
`private`	kein Zugriff möglich	kein Zugriff möglich

Aufrufreihenfolge von Konstruktoren und Destruktoren

Wenn ein Objekt einer abgeleiteten Klasse erzeugt wird, wird nicht nur der Konstruktor der abgeleiteten Klasse aufgerufen, sondern auch der Konstruktor der Basisklasse. Zusätzlich müssen noch die Konstruktoren aller Datenelemente der abgeleiteten Klasse aufgerufen werden. Manchmal hängt die Korrektheit eines Programms von der Reihenfolge dieser Aufrufe ab. Die Konstruktoren werden immer in der folgenden Reihenfolge aufgerufen:

- (1) Der Konstruktor der Basisklasse;
- (2) die Konstruktoren der Datenelemente der abgeleiteten Klasse;
- (3) der Konstruktor der abgeleiteten Klasse selbst.

Falls die Basisklasse wieder eine abgeleitete Klasse ist, setzt sich das Spiel rekursiv fort. Die Destruktoren werden in umgekehrter Reihenfolge aufgerufen. Das folgende Beispiel

verdeutlicht den Mechanismus und zeigt außerdem, daß eine gewöhnliche Elementfunktion, die in der abgeleiteten Klasse neu definiert wird, nur in der aktuellen Version aufgerufen wird.

```
#include <iostream.h>

class Hilfsklasse {
public:
  Hilfsklasse ();
  ~Hilfsklasse ();
};

class Basis {
public:
  Basis ();
  ~Basis ();
  void f ();
};

class Abgeleitet: public Basis {
public:
  Abgeleitet ();
  ~Abgeleitet ();
  void f ();
  Hilfsklasse h;
};

Hilfsklasse::Hilfsklasse () {cout << "Konstr. Hilfsklasse\n";}
Hilfsklasse::~Hilfsklasse (){cout << "Destr.  Hilfsklasse\n";}
Basis::Basis ()             {cout << "Konstr. Basis\n";}
Basis::~Basis ()            {cout << "Destr.  Basis\n";}
void Basis::f ()            {cout << "Fkt.    Basis\n";}
Abgeleitet::Abgeleitet ()   {cout << "Konstr. Abgeleitet\n";}
Abgeleitet::~Abgeleitet ()  {cout << "Destr.  Abgeleitet\n";}
void Abgeleitet::f ()       {cout << "Fkt.    Abgeleitet\n";
}

main () {
  Abgeleitet b;
  b.f ();
}
```

Das Programm erzeugt folgende Ausgabe:

```
Konstr. Basis
Konstr. Hilfsklasse
Konstr. Abgeleitet
Fkt.    Abgeleitet
Destr.  Abgeleitet
Destr.  Hilfsklasse
Destr.  Basis
```

Bei der Erklärung der Aufrufreihenfolge der Konstruktoren sind wir stillschweigend davon ausgegangen, daß die Basisklasse einen Konstruktor ohne Parameter besitzt. Es gibt auch einen speziellen Mechanismus, um Konstruktoren von Basisklassen aufzurufen, die Argumente verlangen, nämlich die *Initialisierer-Liste*. Dazu wird zwischen Funktionskopf und Funktionsblock nach einem Doppelpunkt der explizite Konstruktor-Aufruf angegeben. Im folgenden Prinzip-Beispiel wird vor dem Konstruktor der Klasse `Abgeleitet` mit Hilfe der Initialisierer-Liste `Basis (i)` der Konstruktor der Basisklasse aufgerufen:

```
class Basis {
public:
  Basis (int i);
};

class Abgeleitet: public Basis {
public:
  Abgeleitet ();
};

Basis::Basis (int i) {...}
Abgeleitet::Abgeleitet () : Basis (i) {...}
```

3.9 Virtuelle Funktionen

Späte Bindung

Das alljährliche Brutgeschäft läßt sich für die Klasse aller Vögel in stark vereinfachter Form so beschreiben:

- (1) Sorge für einen geeigneten Nistplatz;
- (2) lege Eier;
- (3) brüte die Eier aus;
- (4) versorge die Jungvögel, bis sie selbständig sind.

Eine schrittweise Zerlegung eines Problems in Teilprobleme ist in der Programmierung nichts neues. Was die Sache hier allerdings interessant macht, ist die Tatsache, daß wir eine Methode für eine Klasse formuliert haben, die auf Hilfsmethoden zugreift, die erst in den Unterklassen definiert werden können. So bedeutet Methode (1) je nach Unterklasse: Höhle zimmern (Spechte), Höhle suchen (Eulen), Nest bauen (Amsel), Wirtsnest aufspüren (Kuckuck). Auch die übrigen Methoden unterscheiden sich in den Unterklassen. So kann bei Methode (3) das Brüten Aufgabe des Weibchens, des Männchens, oder beider sein, oder es kann sogar delegiert werden (Kuckuck).

Man spricht hier von *polymorphen* (vielgestaltigen) Methoden, weil hinter einer Bezeichnung je nach Klasse verschiedene Methoden stecken. Im Alltagsleben wäre eine sinnvolle Kommunikation ohne *Polymorphie* nicht möglich. Sie brauchen nur einmal ein Kochrezept durchzulesen und sich zu überlegen, was die darin enthaltenen Anweisungen für Sie bedeuten, je nachdem, ob Sie der Klasse der Elektroherdbesitzer oder der Gasherdbesitzer angehören, ob Sie eine Küchenmaschine haben und so weiter.

Ähnliches haben wir auch in den Klassen `Kreis` und `Rechteck`: Die Methoden `zeige` und `loesche` sind für beide Klassen unterschiedlich. Dagegen stimmt der Text der Methode `bewege` in beiden Implementierungen überein. Jedes geometrische Objekt kann bewegt werden, indem es zunächst gelöscht und dann mit neuen Koordinaten wieder gezeigt wird. Es bietet sich an, zu den Klassen `Kreis` und `Rechteck` eine Oberklasse `Figur` einzuführen, in der wir unabhängig von der Art der Figur (Kreis oder Rechteck) beschreiben, wie eine Figur bewegt wird:

```
class Figur: public Punkt {    // Basisklasse
```

```
public:                          // für Grafik-Objekte
  Figur (int x0, int y0);
  void bewege (int dx, int dy);
};

void Figur::bewege (int dx, int dy) {
  loesche ();
  x += dx;
  y += dy;
  zeige ();
}
```

Der Haken daran ist nur, daß der Compiler hier streikt, weil er in `Figur::bewege` die Methoden `loesche` und `zeige` gar nicht kennt. Objektorientierte Sprachen bieten einen Ausweg aus dieser Situation. Sie haben C++ bisher als eine typstrenge Sprache kennengelernt. Der Compiler muß bei jedem Objekt wissen, von welchem Typ es ist, um die passenden Methoden einzusetzen. Bei einer Anweisung wie

```
Objekt.zeige
```

prüft der Compiler den Typ der Variablen `Objekt` und setzt dann die Methode `zeige` aus der entsprechenden Klasse ein. Das nennt man *frühe Bindung* (oder auch *statische Bindung*). Von *später Bindung* (oder *dynamischer Bindung*) spricht man dann, wenn der Typ eines Objekts noch nicht vom Compiler erkannt werden kann, sondern erst zur Laufzeit des Programms.

Späte Bindung eröffnet einerseits ungeahnte neue Möglichkeiten, andererseits muß sie mit einem kleinen Verlust an Effizienz erkauft werden. Deshalb kann man in C++ zwischen früher und später Bindung wählen. Zunächst ist die Typstrenge wie folgt aufgelokkert: Einer Variablen vom Typ Zeiger auf eine Klasse darf man einen Zeiger auf eine abgeleitete Klasse zuweisen. Ruft man über diesen Zeiger eine Methode auf, so muß im Normalfall der frühen Bindung der Compiler entscheiden, aus welcher Klasse er die Methode nehmen soll. Dabei muß er sich nach dem Typ der Zeigervariablen richten. Bei später Bindung wird zur Laufzeit der aktuelle Typ des Objekts, auf das der Zeiger zeigt, zugrundegelegt.

Späte Bindung wird in C++ durch *virtuelle Funktionen* realisiert. Durch das Schlüsselwort `virtual` vor der Deklaration einer Elementfunktion wird dem Compiler mitgeteilt, daß er späte Bindung verwenden soll. Technisch wird dies dadurch realisiert, daß jedes Objekt der Klasse zusätzlich einen Zeiger auf eine Tabelle enthält, die zu jeder virtuellen Funktion die zum Objekt gehörende Version angibt.

Ist eine Funktion einmal als virtuell erklärt, ist sie automatisch in jeder abgeleiteten Klasse wieder virtuell. Zum besseren Verständnis sollte man das Schlüsselwort `virtual` bei jeder Neudefinition in einer abgeleiteten Klasse wiederholen, obwohl C++ das freistellt.

In unserem Figuren-Programm werden in der Basisklasse `Figur` die gemeinsamen Eigenschaften der konkreten Klassen `Kreis` und `Rechteck` definiert. Es hat keinen Sinn, Objekte der Basisklasse selbst zu definieren, denn eine allgemeine Figur kann weder gezeigt noch gelöscht werden. Man spricht in solchen Fällen von einer *abstrakten Basisklasse*.

In C++ kann man bei virtuellen Funktionen in einer Basisklasse auf eine Implementierung verzichten, indem man `= 0` hinter die Funktionsdeklaration schreibt. Eine solche Funktion heißt dann *rein virtuell* (pure virtual). Eine Klasse mit mindestens einer rein virtuellen Funktion ist eine abstrakte Basisklasse. Die Definition von Objekten einer solchen Klasse betrachtet der Compiler als Fehler.

In der folgenden Version des Figuren-Programms werden im Hauptprogramm je ein Kreis und ein Rechteck definiert. Einem Zeiger

```
Figur *aktuelleFigur;
```

kann wahlweise die Adresse des Kreises oder des Rechtecks zugewiesen werden. Die Wirkung der Anweisung

```
aktuelleFigur->zeige();
```

kann deshalb erst zur Laufzeit ermittelt werden.

```
////////////////////////////////////////////////////////////////////
//                         FIGUREN.CPP                            //
//          Einfache Demonstration virtueller Funktionen          //
//     am Beispiel der Bewegung von geometrischen Objekten        //
////////////////////////////////////////////////////////////////////

#include <graphics.h>
#include <ctype.h>
#include <conio.h>

class Punkt {
public:
  Punkt (int x0, int y0);
protected:
  int x;
  int y;
};

class Figur: public Punkt {    // abstrakte Basisklasse
public:                        // für Grafik-Objekte
  Figur (int x0, int y0);
  virtual void zeige() = 0;    // rein virtuelle Funktion
  virtual void loesche() = 0;  // dto.
  void bewege (int dx, int dy);
};

class Kreis: public Figur {
public:
  Kreis(int x0, int y0, int r0);
  virtual void zeige();
  virtual void loesche();
protected:
  int Radius;
};

class Rechteck: public Figur {
public:
  Rechteck (int x0, int y0, int x1, int y1);
  virtual void zeige();
  virtual void loesche();
protected:
  int a, b;  // die beiden Seiten
```

```
};

//////////////// Implementierung der Klasse Punkt: ///////////////

Punkt::Punkt(int x0, int y0)  {
  x = x0;  y = y0;
}

//////////////// Implementierung der Klasse Figur: ///////////////

Figur::Figur(int x0, int y0) : Punkt (x0, y0) {}

void Figur::bewege (int dx, int dy) {
  loesche ();
  x += dx;
  y += dy;
  zeige ();
}

//////////////// Implementierung der Klasse Kreis: ///////////////

Kreis::Kreis(int x0, int y0, int r0): Figur (x0, y0) {
  Radius = r0;
}

void Kreis::zeige() {
  circle(x, y, Radius);
}

void Kreis::loesche() {
  setcolor (BLACK);
  circle(x, y, Radius);
  setcolor (WHITE);
}

/////////////// Implementierung der Klasse Rechteck: /////////////

Rechteck::Rechteck (int x0, int y0, int x1, int y1): Figur (x0,y0) {
  a = x1 - x0;
  b = y1 - y0;
}

void Rechteck::zeige() {
  rectangle (x, y, x+a, y+b);
}

void Rechteck::loesche() {
  setcolor (BLACK);
  rectangle (x, y, x+a, y+b);
  setcolor (WHITE);
}

//////////////////////// Test der Klassen: ///////////////////////

int main() {
  char c;
  int graphdriver = DETECT, graphmode;
  initgraph (&graphdriver, &graphmode, "\\borlandc\\bgi");
  outtextxy (10, 8, "Figur mit Cursortasten bewegen. ");
  outtextxy (10,24, "Figur wechseln: [R]echteck  [K]reis");
  outtextxy (10,40, "Ende: [Esc]");
  Kreis K (getmaxx()/2, getmaxy()/2, 40);
  Rechteck R (0, 0, 80, 60);
  Figur *aktuelleFigur = &K;
  aktuelleFigur->zeige();
  do {
    c = getch();      // auf Tastendruck warten
    if (c != 0) {     // normale Zeichentaste: andere Figur?
      aktuelleFigur->loesche ();
```

```
      switch (toupper (c)) {
        case 'K': aktuelleFigur = &K; break;
        case 'R': aktuelleFigur = &R; break;
      }
      aktuelleFigur->zeige ();
    }
    else {            // erweiterter Tastencode: Cursorbewegung?
      c = getch();
      switch (c) {
        case 72: aktuelleFigur->bewege (0, -10); break; // aufwärts
        case 80: aktuelleFigur->bewege (0,  10); break; // abwärts
        case 75: aktuelleFigur->bewege (-10, 0); break; // links
        case 77: aktuelleFigur->bewege (10,  0); break; // rechts
      }
    }
  } while (c != '\33');  // Esc-Taste
  closegraph();
  return 0;
}
```

Übungsaufgabe 13:

Was gibt das folgende Programm aus?

```
#include <iostream.h>

class Klasse1 {
public:
  void frueheBindung ()         {cout << "früheBindung-Klasse1\n";}
  virtual void spaeteBindung () {cout << "späteBindung-Klasse1\n";}
};

class Klasse2: public Klasse1 {
public:
  void frueheBindung ()         {cout << "früheBindung-Klasse2\n";}
  virtual void spaeteBindung () {cout << "späteBindung-Klasse2\n";}
};

void main () {
  Klasse1 Objekt1;
  Klasse2 Objekt2;
  Klasse1 *ptr1, *ptr2;

  cout << "\n";
  ptr1 = &Objekt1;
  ptr2 = &Objekt2;
  ptr1->frueheBindung();
  ptr2->frueheBindung();
  ptr1->spaeteBindung();
  ptr2->spaeteBindung();
}
```

Polymorphe Listen

Eine typische Anwendung der späten Bindung sind *polymorphe Listen*. Ein CAD-Programm muß zum Beispiel eine Vielzahl verschiedener Objekte speichern. Hierzu definiert man eine abstrakte Basisklasse mit virtuellen Funktionen, von der alle benötigen Objekttypen abgeleitet werden können. Alle Objekte können dann in einer Liste von Zeigern auf die Basisklasse verwaltet werden. Eine Bildschirmauffrischung besteht dann einfach darin, daß für alle Elemente der Liste die virtuelle Methode zum Zeichnen aufgerufen wird.

Sie können zum Beispiel das Figuren-Programm mit wenig Aufwand so erweitern, daß es eine Liste (Im einfachsten Fall als Vektor) von Zeigern auf `Figur` verwalten kann.

3.10 Mehrfache Vererbung

Die Art der Vererbung, die Sie bisher kennengelernt haben, würde man in der Biologie als ungeschlechtliche Fortpflanzung bezeichnen: Jede Klasse hat höchstens ein Elternteil, nämlich die Klasse, von der sie abgeleitet ist.

Ebenso wie ein Kind Eigenschaften der Mutter und des Vaters erben kann, ist es in C++ erlaubt, daß eine Klasse von mehreren - sogar beliebig vielen - Klassen abgeleitet wird.

Bei mehrfacher Vererbung müssen bei der Deklaration einer Klasse mehrere Basisklassen angegeben werden. Dabei werden die einzelnen Basisklassen durch Kommata getrennt. Vor der Initialisierung eines Objekts werden die Konstruktoren aller Basisklassen in der Reihenfolge aufgerufen, in der sie bei der Klassendefinition angegeben wurden. Dabei können auch Initialisierer-Listen verwendet werden, in denen die einzelnen Konstruktor-Aufrufe durch Kommata getrennt werden.

Ein einfaches Beispiel

Soll zum Beispiel eine Klasse `Stringfeld` von den Klassen String und Position abgeleitet werden, sieht das so aus:

```
class Stringfeld: public String, public Position {
  ...
```

Es folgt ein einfaches (und damit praxisfernes) Beispiel. Ein Stringfeld hat einen Inhalt und eine Position auf dem Bildschirm, die in zwei getrennten Basisklassen beschrieben werden:

```
#include <string.h>
#include <conio.h>

class String {
public:
  String (char *s);
protected:
  char str[80];
};

String::String (char *s) {
  if (strlen (s) < 80) strcpy (str, s);
}

class Position {
public:
  Position (int x, int y);
protected:
  int x0, y0;
};

Position::Position (int x, int y) {
  x0 = x;
```

```
  y0 = y;
}

class Stringfeld: public String, public Position {
public:
  Stringfeld (char *s, int x, int y);
  void zeige ();
};

Stringfeld::Stringfeld (char *s, int x, int y) :
  String (s), Position (x, y) {
  zeige ();
}

void Stringfeld::zeige () {gotoxy (x0,y0); cputs (str);}

int main () {
  Stringfeld s ("Stringfeld", 35, 13);
  return 0;
}
```

Dieses Programm gibt die Zeichenkette "`Stringfeld`" etwa in der Mitte des Bildschirm aus. Ein sinnvolles Beispiel für mehrfache Vererbung finden Sie zum Beispiel im Modul `EDT` des Projekts `DIAGRAMM` auf der Diskette zum Buch.

Virtuelle Basisklassen

Ein Problem bei mehrfacher Vererbung ist die "Inzucht". Wenn eine Klasse von mehreren Basisklassen abgeleitet wird, die wiederum eine gemeinsame Basisklasse haben, gibt es die Elemente dieser gemeinsamen Basisklasse mehrfach. Das führt zu Mehrdeutigkeiten und erhöhtem Speicherbedarf:

```
class Mutter: public Ahn {...};
class Vater: public Ahn {...};
class Kind: public Vater, public Mutter {...}
```

In diesem Beispiel hat jedes Objekt der Klasse `Kind` die aus der Klasse `Ahn` geerbten Elemente doppelt. Um das zu vermeiden, läßt sich die Basisklasse als *virtuelle Basisklasse* deklarieren:

```
class Mutter: virtual public Ahn {...};
class Vater: virtual public Ahn {...};
class Kind: public Vater, public Mutter {...}
```

Hier wurde `Ahn` zu einer virtuellen Basiklasse und gibt seine Elemente nur einmal an `Kind` weiter.

3.11 Dynamische Objekte als Funktionswert

Ein Ziel von C++ ist es, vom Anwender definierten Datentypen die gleichen Möglichkeiten wie Standard-Datentypen geben zu können. In der Praxis tauchen dabei allerdings einige technische Probleme auf, die nicht ganz einfach zu umgehen sind.

Wir wollen nochmals einen Datentyp string entwerfen, diesmal aber nicht nach dem Vorbild von Turbo Pascal, sondern so flexibel, daß ein String im Prinzip jede beliebige Größe annehmen kann. Für diese Klasse wollen wir die Verkettung mit dem Operator + definieren. Sind s1, s2 und s3 vom Typ string, so soll die Zuweisung

```
s1 = s2 + s3;
```

die Strings s2 und s3 verketten und das Ergebnis der Stringvariablen s1 zuweisen. Das soll auch dann klappen, wenn s1 derzeit zu kurz ist.

Dazu könnten wir so vorgehen: Die Operatorfunktion + gibt den gewünschten verketteten String zurück. Der Zuweisungsoperator wird so überladen, daß er zunächst das aktuelle Objekt s1 löscht (es hat ja meist eine nicht passende Größe) und es dann mit dem Kopier-Konstruktor neu erzeugt. Diese Vorgehensweise ist allerdings extrem ineffizient. Um das zu verstehen, wollen wir etwas genauer betrachten, wie der Compiler die Zuweisung eines Funktionswertes handhabt.

Zuweisung eines Funktionswertes

Betrachten Sie folgendes Programmfragment:

```
int f (int a, int b) {
  int t = a + b;
  return t;
}

main () {
  int a, b, c;
  c = f (a, b);
}
```

Was geschieht in der letzten Programmzeile? Zunächst wird die Funktion f aufgerufen. Diese erzeugt das lokale Objekt t und gibt es mit der return-Anweisung zurück. Anschließend wird der Zuweisungsoperator aufgerufen. Aber jetzt existiert die lokale Variable t gar nicht mehr. Aus diesem Grund wird in der return-Anweisung ein verborgenes temporäres Objekt erzeugt, das beim Aufruf des Zuweisungsoperators noch existiert und irgendwann später gelöscht wird (Wann das genau geschieht, ist vom Compiler abhängig).

Referenzenzähler

Bei unserem String-Beispiel wird die Operator-Funktion + aufgerufen und erzeugt ein temporäres Objekt. Dieses wird beim Verlassen der Funktion in eine verborgene temporäre Variable kopiert, und diese wird schließlich in die Zielvariable kopiert. Stellen Sie sich vor, Sie möchten zwei 100-mal-100-Matrizen addieren. Dann bestünde die Hauptarbeit des Programms im zweimaligen Kopieren des errechneten Ergebnisses. In der Praxis würden Sie dann vermutlich auf die Eleganz einer Zuweisung der Form

```
C = A + B;
```

verzichten und sich eine Funktion schreiben, die die Multiplikation ohne unnötige Kopien erreicht. Das würde dann vielleicht so aussehen:

```
addiere (A, B, C);
```

Es gibt doch eine Möglichkeit, die Eleganz mit der Effizienz zu paaren, allerdings mit einigen formalen Verrenkungen. Der Schlüssel zur Lösung liegt darin, daß wir die eigentlichen (und speicheraufwendigen) Daten vom Objekt trennen: Wir zerlegen den String in ein Objekt vom Typ `string` und ein Objekt vom Typ `stringDaten`. Ein `string` enthält jetzt nur noch einen Zeiger auf seine `stringDaten`. So ist es möglich, daß mehrere gleiche `string`-Objekte auf die gleichen `stringDaten` zeigen. Bei dieser Lösung müssen wir nur folgendes beachten:

- Bei einer *Zuweisung* an einen `string` muß geprüft werden, ob er seine `stringDaten` mit anderen `string`-Objekten teilt. Hierfür bekommt die Klasse `stringDaten` ein zusätzliches Element `Referenzen`, das zählt, wieviele `string`-Objekte derzeit auf ein `stringDaten`-Objekt zeigen. Ist er der einzige (`Referenzen == 1`), ruft er den Destruktor seiner `stringDaten` auf und hängt sich an die `stringDaten` des zugewiesenen Strings. Ist `Referenzen` > 1, wird der Zähler nur um eins vermindert.
- Der *Destruktor* eines Strings vermindert den Referenzenzähler seiner `stringDaten` um eins und ruft den Destruktor der `stringDaten` nur dann auf, wenn der Referenzenzähler Null erreicht hat.

Implementierung der Klassen stringInfo und string

In der Implementierung des Operators + wird zunächst eine lokale Variable `t` der passenden Länge erzeugt. Der Referenzenzähler steht jetzt auf 1. Bei der Übergabe des Funktionswertes wird er in die verborgene temporäre Variable kopiert. Jetzt steht der Referenzenzähler auf 2. Bevor die Zuweisung ausgeführt wird, wird der Destruktor der lokalen Variable `t` aufgerufen. Dabei werden die `stringDaten` nicht vernichtet, sondern nur der Referenzenzähler von 2 auf 1 herabgesetzt.

```
//////////////////////////// STRING.CPP ////////////////////////////
//                                                                 //
//            Verwendung von Referenzenzählern                     //
//             zur Vermeidung unnötiger Kopien                     //
//                                                                 //
/////////////////////////////////////////////////////////////////////
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

struct stringDaten {
  char *s;
  int  Referenzen;
};

class string {
public:
  string (int);                     // string s (256);
  string (const char *);            // string s = "abc";
```

```
  string (const string&);             // string t = s;
  ~string ();
  int length ();
  string& operator = (const string&); // t = s;
  char& operator [] (int i);
  friend ostream& operator << (ostream&, string&);
  friend string operator + (const string& s, const string& t);
  friend int operator == (const string& s, const string& t);
  friend int operator != (const string& s, const string& t);
private:
  stringDaten *p;
};

void error (char *p) {
  cerr << p << "\n";
  exit (1);
}

string::string (int n) {
  p = new stringDaten;
  p->s = new char [n+1];
  p->Referenzen = 1;
}

string::string (const char *s) {
  p = new stringDaten;
  p->s = new char [strlen(s)+1];
  strcpy (p->s, s);
  p->Referenzen = 1;
}

string::string (const string& s) {
  s.p->Referenzen++;
  p = s.p;
}

string::~string () {
  if (--p->Referenzen == 0) {
    delete p->s;
    delete p;
  }
}

inline int string::length () {return strlen (p->s);}

string& string::operator = (const string& s) {
  if (--p->Referenzen == 0) {
    delete p->s;
    delete p;
  }
  s.p->Referenzen++;
  p = s.p;
  return *this;
}

char& string::operator [] (int i) {
  if (i<0 || i >= strlen(p->s)) error ("unerlaubter Index");
  return p->s[i];
}

int operator == (const string& s, const string& t) {
  return strcmp (s.p->s, t.p->s) == 0;
}

int operator != (const string& s, const string& t) {
  return strcmp (s.p->s, t.p->s) != 0;
}

ostream& operator << (ostream& s, string& t) {
```

```
  return s << t.p->s << t.p->Referenzen << " ";
}

string operator + (const string& s1, const string& s2) {
  int l1 = strlen (s1.p->s);
  string t (l1 + strlen(s2.p->s));
  strcpy (t.p->s, s1.p->s);
  strcpy (t.p->s+l1, s2.p->s);
  return t;
}

main () {
  string s1 = "erster";
  string s2 = "zweiter";
  string s3 = "dritter";
  s3 = s1 + s2;
  cout << s1 << s2 << s3 << "\n";
}
```

Nun haben Sie die nötigen Kenntnisse, um ähnliche Aufgaben, wie etwa Matrizenrechnung elegant zu lösen. Hierzu soll ein Hinweis genügen: Sie können eine *Matrix* als Vektor von Zeilenvektoren definieren. Ist `M` eine Matrix, dann sollte der Operator `[]` für `M[i]` eine Referenz auf den `i`-ten Zeilenvektor von `M` liefern. Dann ist automatisch `M[i][k]` eine Referenz auf das entsprechende Matrix-Element.

Anhang

B Lösungen der Übungsaufgaben

Aufgabe 1:

```
if (a > 0 && a <= 4)
  b = a;
```

Aufgabe 2:

Die `for`-Schleife wird bei `i=21` mit der `break`-Anweisung abgebrochen. Für `i=0` .. `20` (also 21 mal) wird `n` um 100 erhöht. Die Zeile `n++` wird erst ab `i=10` (also 11 mal) ausgeführt. Damit bekommt `n` den Wert 21*100+11 = 2111.

Aufgabe 3:

Für die nicht angegebenen Parameter wird die kleinstmögliche `int`-Zahl eingesetzt. Sie ist in der Header-Datei `LIMITS.H` definiert:

```
#include <limits.h>
int max0 (int a, int b = INT_MIN, int c = INT_MIN) {
  if (a > b)
    if (a > c) return a;
    else return c;
  else
    if (b > c) return b;
```

```
    else return c;
}
```

Es folgt eine alternative Lösung mit einem bedingten Ausdruck:

```
int max (int a, int b = INT_MIN, int c = INT_MIN) {
  return (a > b) ? ((a > c) ? a : c) : ((b > c) ? b : c);
}
```

Aufgabe 4:

```
  f (i,l);         // 'A'
  f (l,i);         // 'B'
  f (i,i);         // Fehler: mehrdeutig!
  f (l,l);         // Fehler: mehrdeutig!
  f (i, long(i)); // 'A'
  f (l, int(i));  // 'B'
```

Aufgabe 5:

In der einfachen Zuweisung wird der mit allen Zeigertypen verträgliche `NULL`-Zeiger zugewiesen. Die doppelte Zuweisung ist so zu lesen:

```
  p1 = (p2 = NULL);
```

Dabei ist der Klammerausdruck vom Typ `long *` und nicht typverträglich mit `int *`.

Aufgabe 6:

Ab der Adresse `s` werden alle Bytes mit `'V'` beschrieben. (Systemzusammenbruch ?!)

Aufgabe 7:

Der Compiler reserviert den Speicherplatz für die drei Strings unmittelbar hintereinander. Deshalb können die vorderen Strings in die hinteren überlaufen. Das Diagramm zeigt den Speicherbereich Schritt für Schritt. Dabei steht ein Backslash für das Null-Byte.

```
s1    s2  s3
³.....³...³...
³.....³...BAR\
³.....WIRKLICH\
WUNDER\IRKLICH\
WUNDERLICH
```

Die Lösung ist also `"WUNDERLICH"`.

Aufgabe 8:

```
A:e, B:g, C:h, D:f, E:c, F:b, G:d, H:a.
```

Aufgabe 9:

```
int (*(* A[n])(int))[n];
 6   4 2 0 1    3    5
```

(0) A ist (1) Vektor[n] von (2) Zeigern auf (3) Funktion (int) mit Wert (4) Zeiger auf (5) Vektor[n] von (6) int.

```
typedef int Typ5[n];      // Typ5 ist Vektor[n] von int
typedef Typ5 *Typ4;       // Typ4 ist Zeiger auf Typ5
typedef Typ4 Typ3 (int); // Typ3 ist Funktion (int) mit Wert Typ4
typedef Typ3 *Typ2;       // Typ2 ist Zeiger auf Typ3
typedef Typ2 Typ1[n];     // Typ1 ist Vektor[n] von Typ2
Typ1 a;
```

Aufgabe 10:

"`Quadrat (b+c)`" wird zu "`b+c*b+c`" erweitert (was gleich `b + (c*b) + c` ist). `Doppelt` ist ein Makro ohne Parameter, weil nach dem Namen ein Leerzeichen folgt! Deshalb wird "`Doppelt (b+c)`" zu "`(x) x*x (b+c)`" erweitert (Syntaxfehler).

Ersatz:

```
inline Quadrat (int x) {return x*x;}
inline Doppelt (int x) {return x+x;}
```

Aufgabe 11:

Die Klasse `bitset` wird erweitert:

```
class bitset {
public:
  ...
  bitset (int i);            // Konversion int --> bitset
  operator int ();           // Konversion bitset --> int
  ...
};
```

Implementierung:

```
inline bitset::bitset (int i)           {bits = i;}
inline bitset::operator int ()          {return bits;}
```

Aufgabe 12:

Wenn der Gleichheitsoperator nicht überladen wird, verwendet der Compiler die implizite Konvertierung nach `double`. Dabei kann das Resultat aber durch Rundungsfehler verfälscht werden. Erst, wenn Sie auch die Typumwandlung (`operator double`) entfernen, gibt der Compiler eine Fehlermeldung aus.

Aufgabe 13:

`ptr1` und `ptr2` sind vom gleichen Typ Zeiger auf `Klasse1`. Daher setzt der Compiler in den beiden Anweisungen

```
  ptr1->frueheBindung();
  ptr2->frueheBindung();
```

die Funktion der Klasse 1 ein. In den beiden Anweisungen

```
  ptr1->spaeteBindung();
  ptr2->spaeteBindung();
```

wird eine virtuelle Funktion aufgerufen. Deshalb werden (zur Laufzeit) die Funktionsversionen nach dem Typ des Objekts, auf das der Zeiger zeigt, ermittelt. Insgesamt werden so die vier Zeilen ausgegeben:

```
früheBindung-Klasse1
früheBindung-Klasse1
späteBindung-Klasse1
späteBindung-Klasse2
```

C Mehrdeutige Sprachelemente

C++ hat zwar auf den ersten Blick einen recht kleinen Sprachumfang. Viele Schlüsselwörter und Operatoren haben allerdings in verschiedenen Kontexten unterschiedliche Bedeutungen. Hier finden Sie eine Übersicht mehrdeutiger Sprachelemente, die Ursache von Programmierfehlern sein können:

const

(1) `const`-Variablen: Können nicht verändert werden.

(2) `const`-Zeiger: Nicht der Zeiger ist konstant, sonden das Objekt, das er adressiert.

(3) `const` Elementfunktion

friend

(1) befreundete Klasse (darf auf private Elemente zugreifen).

(2) befreundete Member-Funktion

operator

(1) Überladen von Operatoren (member/friend)

(2) implizite Typkonvertierung

static

(1) Funktionen und globale Variablen: Sichtbarkeit auf die Datei beschränkt.

(2) lokale Variablen: Lebensdauer auf Programmlaufzeit ausgedehnt.

(3) Klassenelemente: Nur einmal für eine Klasse vorhanden.

virtual

(1) virtuelle Funktion

(2) virtuelle Basisklasse (bei Mehrfachvererbung)

void

(1) leerer Ergebnistyp von Funktionen: `void gotoxy (int x, int y)`

(2) Bezeichnung einer leeren Parameterliste, z. B.: `int rand (void)`
Diese Verwendung ist in C++ überflüssig. Sie ist in ANSI-C nötig, um klarzumachen, daß keine Parameter vergessen worden sind.

(3) `void *` bezeichnet universellen Zeiger.

(4) In Zortech C++: leere Anweisung. Hiermit wird angedeutet, daß wirklich eine leere Anweisung gewünscht wird. Beispiel `while (! keypressed ()) void;` Der Zortech-Compiler gibt hier eine Warnung aus, wenn `void` weggelassen wird.

while

(1) Vorbedingung der `while`-Schleife

(2) Nachbedingung in do-Schleife

Operator ,

(1) Trenner zwischen Funktionsparametern

(2) Komma-Operator

Operator ()

(1) Funktionsaufruf bzw. Parameterliste

(2) Typ-Umwandlung: **Typname (Ausdruck)**

(3) Cast-Operator: **(Typ) Ausdruck**

Operator *

(1) Binär: Multiplikationsoperator

(2) Unär, präfix: Dereferenzierungsoperator (auch zur Zeigerdefinition)

Operator &

(1) Unär, präfix: Adreß-Operator

(2) Unär, postfix: Referenz-Operator

(3) Binär: Bitweise Und-Verknüpfung

Operator ~

(1) Bitweise Negation

(2) Bezeichnung des Destruktors

Operator ::

(1) Binär: Klassenspezifikation

(2) Unär: Ansprechen globaler Objekte

D Unterschiede zwischen C++ und C

C++ ist eine Fortentwicklung der Sprache C. Das heißt aber nicht, daß jedes C-Programm automatisch auch mit dem C++-Compiler übersetzt werden kann. Die Kompatibilität besteht vielmehr darin, daß C-Moduln und C++-Moduln beliebig zu lauffähigen Programmen gebunden werden können.

C selbst hat mehrere Entwicklungsstufen durchlaufen. In der ursprünglichen Version vor der ANSI-Normierung gab es keinerlei Überprüfung von Funktionsaufrufen auf korrekte Parameter- und Rückgabetypen. C war eine Sprache von Freaks für Freaks. Was machte es da schon, daß ein Programm vom Compiler klaglos übersetzt wurde, dann beim Lauf abstürzte und ein langwieriges Suchspiel begann. Der Pascal-Entwickler Niklaus Wirth hat kürzlich wieder in einem Interview (iX 5/1991) die Sprache C in Grund und Boden verdammt: "Natürlich weiß ich auch, daß man mit der schönsten Programmiersprache scheußliche Programme konzipieren kann. Da gibt es leider so viele Beispiele. Mit C ist es

weitaus schlimmer: man kann einfach alles damit machen, wenn man gerissen ist. Da wird dann herumgetrickst, ganz schrecklich ist das."

Während C durch die ANSI-Norm wohl nun eine stabile Sprachdefiniton ist, wird C++ sich durchaus noch weiterentwickeln. Der in diesem Buch behandelte Sprachstandard (wie er in Borland C++ implementiert ist) entspricht der 1989 erschienenen Version 2.0 des Sprachentwicklers (AT&T). In früheren Versionen gab es noch keine mehrfache Vererbung. Auch wurde die Typstrenge etwas lockerer gehandhabt.

C++ hat zwar viele Mängel von C abgelegt. Trotzdem ist es wohl immer noch eine gefährliche Programmiersprache. Das liegt einerseits daran, daß viele Sprachkonzepte etwas undurchsichtig und unsystematisch entworfen sind, andererseits an den vielen überflüssigen Freiheiten, die einen einheitlichen Programmierstil geradezu torpedieren. Dieses Buch sollte ein wenig dazu beitragen, die in C++ verborgenen Gefahren auf ein erträgliches Maß zu begrenzen.

E Rangfolge der Operatoren

Op.	Bezeichnung	Anwendung
::	Klassenspezifikation	Klasse::Element
::	globales Objekt	::Name
.	Elementauswahl	Record.Element
->	dto. mit Dereferenzierung	Zeiger->Element
[]	Vektorelement	Zeiger[int-Ausdruck]
()	Funktionsaufruf	Fkt-Ausdr. (Parameterliste)
()	Typumwandlung	Typ (Ausdruck)
sizeof	Größe	sizeof (Ausdruck)
sizeof	Größe	sizeof (Typ)
++	Prä-Inkrementierung	++Lvalue
++	Post-Inkrementierung	Lvalue++
--	Prä-Dekrementierung	--Lvalue
--	Post-Dekrementierung	Lvalue--
~	bitweise Negation	~ Ausdruck
!	logische Negation	! Ausdruck
+	unäres Plus	+ Ausdruck
-	unäres Minus	- Ausdruck
&	Adreßoperator	& Lvalue
*	Dereferenzierung	* Ausdruck
new	Speicherreservierung	new Typ
new	Speicherreservierung	new Typ [int-Ausdruck]
delete	Speicherfreigabe	delete Zeiger
delete	Speicherfreigabe	delete [int-Ausdruck] Zeiger
()	Typkonvertierung (Cast)	(Typ) Ausdruck

->* Auswahloperationen für Zeiger (in
.* diesem Buch nicht behandelt)

---

* Multiplikation Ausdruck * Ausdruck
/ Division Ausdruck / Ausdruck
% Rest (Modulo) Ausdruck % Ausdruck

+ Addition Ausdruck + Ausdruck
- Subtraktion Ausdruck - Ausdruck

<< bitweises Linksschieben Ausdruck << Ausdruck
>> bitweises Rechtsschieben Ausdruck >> Ausdruck

< kleiner als Ausdruck < Ausdruck
> größer als Ausdruck > Ausdruck
<= kleiner als oder gleich Ausdruck <= Ausdruck
>= größer als oder gleich Ausdruck >= Ausdruck

== Gleichheit Ausdruck == Ausdruck
!= Ungleichheit Ausdruck != Ausdruck

& bitweises Und Ausdruck & Ausdruck

---

^ bitweises exklusives Oder Ausdruck ^ Ausdruck

| bitweises Oder Ausdruck | Ausdruck

&& logisches Und Ausdruck && Ausdruck

|| logisches Oder Ausdruck || Ausdruck

? : bedingter Ausdruck Ausdruck ? Ausdruck : Ausdruck

= Zuweisung Lvalue = Ausdruck
*= Zuweisung mit Multiplikation Lvalue *= Ausdruck
/= Zuweisung mit Division Lvalue /= Ausdruck
%= Zuweisung mit Rest Lvalue %= Ausdruck

+= Zuweisung mit Addition Lvalue += Ausdruck

-= Zuweisung mit Subtraktion Lvalue -= Ausdruck

<<= Zuweisung mit Linksschieben Lvalue <<= Ausdruck

>>= Zuweisung mit Rechtsschieben Lvalue >>= Ausdruck

&= Zuweisung mit bitweisem Und Lvalue &= Ausdruck

|= Zuweisung mit bitweisem Oder Lvalue |= Ausdruck

^= Zuweisung mit bitweisem exkl. Oder Lvalue ^= Ausdruck

, Sequenzoperator Ausdruck , Ausdruck

Literaturverzeichnis

[Gamm95] **E. Gamma, R. Helm, R. Johnson, J. Vlissides:** *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, 1995.

[Kern88] **B. W. Kernighan, D. M. Ritchie:** *The C Programming Language*, 2nd Ed., Prentice-Hall, Englewood Cliffs, N.J., 1988.

[Lipp89] **S. B. Lippmann:** *A C++ Primer*, Addison-Wesley, Reading, 1989. Deutsche Übersetzung: *C++ Einführung und Leitfaden*, Addison-Wesley, Bonn, 1990.

[Louis96] **D. Louis:** *C und C++. Programmierung und Referenz*, Markt und Technik, 1996.

[Meye88] **B. Meyer:** *Object-oriented Software Construction*, Prentice Hall, New York 1988. Deutsche Übersetzung: *Objektorientierte Software-Entwicklung*, Hanser, München, 1990.

[Scha95] **M.Schader, S. Kuhlins:** *Programmieren in C++. Einführung in den Sprachstandard C++*, Springer, Berlin, 3. Aufl. 1995.

[Seth89] **R. Sethi:** *Programming Languages. Concepts and Constructs*, Addison-Wesley, Reading, 1989.

[Stro91] **B. Stroustrup:** *The C++ Programming Language*, Bell Telephone Laboratories, Incorporated, 2. Aufl. 1991 (Reprinted with corrections December, 1994).
Deutsche Übersetzung: *Die C++ Programmiersprache*, Addison-Wesley, Bonn, 2. Aufl. (4. überarbeiteter Nachdruck) 1994.

[Watt90] **D. A. Watt:** *Programming Language Concepts and Paradigms*, Prentice Hall, New York, 1990

[Wien88] **R. Wiener, L. Pinson:** *An Introduction to Object-Oriented Programming and C++*, Addison-Wesley, Reading, 1988.

Zurück